

Inside Ferrous Lineup

Ferrous Lineup contributors

Contents

Introduction	5
Building it from source	8
Part I: The shape of the program	
Five crates, and what each one may know	12
Two threads: the window and the audio	22
The project: patterns, channels, tracks, clips and markers	36
Telling the audio thread what changed	51
Undo and redo	69
Saving: the .lineup file, recovery and damaged files	78
Drawing the windows with egui: layout, zoom, themes, icons	98
The editors: channel rack, piano roll and playlist	126
Two languages: translations, and letters or solfège	164
Exercises: the shape of the program	175
Part II: Sound	
The sequencer: timing to the sample	189
Tempo, meter, swing and feel	202
Instruments: drums, sampler and Lineup Osc	228
SoundFonts and General MIDI	246
The mixer: levels, pan, reverb, effects and score order	265
MIDI: messages, devices, recording and files	285
Exporting: WAV, MIDI and video	337
The playback trace	369
Exercises: sound	383
Part III: Music theory in code	
Pitch classes, notes and spelling	398
Scales, modes and keys	412
Chords: building, naming and inversions	429
Roman numerals, the harmony keys and chord functions	449
Editing tools as plain functions	468

Hearing harmony: chords and keys from notes	481
Pitch-class sets, and one harmony with four names	501
Twelve-tone rows and their rules	516
Micromodes	543
Voice leading and part motion	563
Importing a recording (experimental)	575
Exercises: music theory in code	597
Part IV: The band	
The library: styles and rhythms as data	613
The chord sheet: model, layout and text	633
How the band plays a chord sheet	668
An orchestra for music that changes meter	700
Song form, sections, variations and locks	715
Shots, holds, rests and pushes	736
The soloist	746
Your own styles: the style editor	760
Practice tools: metronome, speed trainer, ear training	770
The lead sheet: from notes to notation	798
Exercises: the band	820
Part V: Songs as data	
How a demo is written	837
From a MIDI performance or a score to a demo	853
Exercises: songs as data	862
Part VI: Building, testing and shipping	
How the code is tested: unit tests, window tests, traces	868
Three systems and continuous integration	873
Release packages	880
How the books are built	892
Reference	
Project file format	899
Rust glossary	905

Music glossary 920

Ideas for what comes next 925

Index 926

Introduction

Ferrous Lineup is an open source program, written in Rust, for playing songs with a **backing band**. Type the chords of a song bar by bar, pick a style, and a band (bass, drums and piano, with a guitar, strings, horns or an organ when the style has them) plays it in any key and tempo. Every part the band plays lands on a track of its own, where you can open it in the piano roll, change it by hand, or leave it out and play it yourself.

It is also a way to **learn harmony** while you play. It names every chord you hear, with its numeral in the key and what it does there, and it knows the other ways music names harmony too: pitch-class sets, twelve-tone rows and Kröpfl's micromodes.

This book is about the inside: **how the code is organized, how it is designed and why**, and how the music theory the app knows is written in Rust. It explains the Rust each part uses as it goes, pointing at the real files, so it is also a Rust course built around a real program. You do not need to know Rust before reading; if you know another programming language, you can follow along. To *use* the app, read the [user manual](#) instead: it has no code at all.

How the book is organized

The chapters follow the code, from the program's shape to how it ships. Each says what a part of the app does, with a link to the manual page on using it; how it is designed and why; the Rust it uses; the files it lives in; and the tests that guard it. Exercises close each part.

Part	About	Rust along the way
I. The shape of the program	the five crates, the two threads, the project's data, undo, saving, the windows, two languages	workspaces and crates, modules, enums with data, rules kept by one gatekeeper, <code>Arc</code> and a lock-free ring buffer, <code>History<T></code> , <code>serde</code> , error enums, <code>Drop</code> , immediate-mode drawing, thread-local settings
II. Sound	the sequencer, tempo and meter, the instruments, <code>SoundFonts</code> , the mixer, MIDI,	timing to the sample, atomics, <code>FnMut</code> closures, a <code>Mutex</code> and <code>mpsc</code> queues, byte strings and

Part	About	Rust along the way
	exporting, the playback trace	slice patterns, the one <code>unsafe</code> block, <code>Copy</code> types with rules
III. Music theory in code	pitch, scales and keys, chords, numerals and functions, editing tools, hearing harmony, sets, rows, micromodes, voice leading, importing a recording	<code>match</code> tables, bit sets, iterators (<code>find_map</code> , <code>cycle</code> , <code>chunk_by</code>), collecting <code>Options</code> , <code>FromStr</code> and <code>Display</code> , <code>try_from</code> , dynamic programming
IV. The band	the library, the chord sheet, how the band plays, the orchestra, song form, stops, the soloist, your own styles, practice tools, the lead sheet	<code>static</code> tables and <code>const fn</code> , <code>macro_rules!</code> , hashing and seeded generators, parsing text, drawing a PDF
V. Songs as data	how a demo is written, and how a MIDI performance or a score becomes one	<code>const</code> data in child modules, struct update syntax, <code>LazyLock</code> , sorting with <code>Ordering</code>
VI. Building, testing and shipping	the tests, the three systems and their CI, release packages, how these books are built	<code>#[cfg(test)]</code> , integration tests, <code>catch_unwind</code> , <code>cfg</code> and features, build scripts

At the back are the [project file format](#), a [Rust glossary](#), a [music glossary](#) and [ideas for what comes next](#).

Where to start

- **To learn Rust:** read the parts in order, with the [Rust glossary](#) open, and try the exercises at the end of each part. [Building it from source](#) gets the code running first.
- **To learn how the music theory is programmed:** start with Part III, and keep the [music glossary](#) close. The theory itself, without code, is taught in the manual.
- **To change something:** find the part of the app in the contents, read its chapter, and run its tests before and after your change.

How to read it

1. Online, at <https://naelolaiz.github.io/ferrous-lineup/>.
2. As one PDF, to read offline or print: [ferrous-lineup-book.pdf](#). It has page numbers, the contents at the front, an index of words at the back, and bookmarks for every chapter.
3. As Markdown files, in the repository's `book/src` folder, or as a website on your computer with `mdbook serve --open` in `book/`.

The book is also in Spanish, at <https://naelolaiz.github.io/ferrous-lineup/es/>. [How the books are built](#) tells how the websites and PDFs are made.

Code links point at the `main` branch, and the snippets in the chapters are copied from it. Where a snippet leaves lines out, it says so with `// ...`.

Name and license

“Ferrous” means “made of iron”, and rust is iron oxide; a lineup is the players of a band, the band that plays your songs.

Ferrous Lineup is free software, released under the GNU General Public License, version 3 or later.

Building it from source

To use Ferrous Lineup, the ready-made downloads are enough (see the manual's [Installing it](#)). To read its code with this book, change it and run its tests, build it yourself. It takes a few commands on Linux, macOS or Windows.

1. Install Rust

Rust is installed with **rustup**, a small tool that downloads the compiler (`rustc`) and the build tool (`cargo`) and keeps them up to date:

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

On Windows, download and run `rustup-init.exe` from <https://rustup.rs> instead. Restart the terminal afterwards and check that it worked:

```
cargo --version  
rustc --version
```

Ferrous Lineup needs Rust 1.95 or newer, as `rust-version` in the root `Cargo.toml` says: it uses the 2024 edition of the language (see [Editions](#)), `if let` chains, which arrived in Rust 1.88, and file locks, which arrived in 1.89, and egui and eframe, which draw the window, need 1.95. If your Rust is older, `rustup update` fixes it.

2. Install the system's build tools

Rust downloads its own libraries, the *crates*, by itself, but it needs a linker from your system, and on Linux a few crates talk to libraries that come with the system: ALSA for sound, and X11 or Wayland for windows.

Debian or Ubuntu:

```
sudo apt install build-essential pkg-config libasound2-dev \  
libxkbcommon-dev libxkbcommon-x11-0 libwayland-dev
```

Fedora:

```
sudo dnf install gcc pkgconf-pkg-config alsa-lib-devel \
    libxkbcommon-devel libxkbcommon-x11 wayland-devel
```

File dialogs use the desktop's portal, which GNOME and KDE already have; on a bare window manager, install `xdg-desktop-portal-gtk`. MIDI goes through the ALSA *sequencer*, which desktop systems load already; if the app's MIDI menu says MIDI is unavailable, load it with `sudo modprobe snd-seq`.

macOS: Apple's command line tools bring the linker and the system libraries:

```
xcode-select --install
```

Windows: when `rustup-init.exe` asks, let it install the Visual Studio Build Tools; Rust uses their linker. Then use the commands below in PowerShell or the Command Prompt.

3. Get the code and run it

Clone the repository (Git comes with the build tools on macOS and Linux; on Windows, install it from <https://git-scm.com>) and build:

```
git clone https://github.com/naelolaiz/ferrous-lineup
cd ferrous-lineup
cargo run --release
```

The first build takes a few minutes, because Cargo compiles every dependency; later builds compile only what changed. `--release` turns on the optimizations, which audio code needs to keep up. While you work on the code, `cargo run` alone builds faster, and keeps the console where messages and panics show.



The first time, a window over it offers real instrument sounds; [SoundFonts and General MIDI](#) says what it does.

The manual's [The main window](#) says what each part of it does.

4. From the command line

The program takes a few arguments of its own. After `cargo run`, `--` says that what follows is for the program, not for Cargo:

```
cargo run --release -- my-song.lineup # open a project
cargo run --release -- --demos # list the demos
cargo run --release -- --trace "Frère Jacques" --out round.tsv # play a demo headless, note by note
cargo run --release -- --help
```

`--trace` is the [playback trace](#): it plays a project or a demo the way **File > Export WAV...** does, without a window or a sound card, and writes every note it starts to a table.

5. Run the tests

```
cargo test
```

runs every test of every crate of the workspace. [How the code is tested](#) says what kinds there are and how to run some of them.

6. Build the books

The books are written with [mdBook](#), a Rust tool. Install it once, at the version the workflow uses, then build:

```
cargo install mdbook --version 0.5.4 --locked
cd book
mdbook serve --open
```

`mdbook serve` builds the website, opens it, and builds it again each time you save a page. `book-es/`, `manual/` and `manual-es/` work the same way. [How the books are built](#) also prints the PDF.

Useful commands

Command	What it does
<code>cargo build</code>	Compile everything, in debug mode
<code>cargo run</code>	Compile and start the app
<code>cargo test</code>	Run all the tests
<code>cargo fmt</code>	Format all the code in the standard Rust style
<code>cargo clippy</code>	Run the linter, which suggests better ways to write things
<code>cargo doc --open</code>	Build the API documentation from the <code>///</code> comments and open it

Five crates, and what each one may know

Files: `Cargo.toml`, the `Cargo.toml` of each crate in `crates/`, and `crates/lineup-core/src/lib.rs`

Ferrous Lineup is one program, `ferrous-lineup`, built from five Rust crates. This chapter is about how they are split and why: what Cargo is, what a crate, a workspace and a module are, a first small type and the Rust it shows, and the few rules the whole program keeps, each explained in a chapter of its own later in the book. The last section is a map of where each part of the app lives.

Cargo

Cargo is Rust's build tool and package manager in one. It reads `Cargo.toml` files, downloads dependencies from crates.io, compiles everything in the right order, runs tests and much more. You will almost never call the compiler `rustc` directly.

Crates

A **crate** is Rust's unit of compilation: one library or one program. Ferrous Lineup is five crates, each with a job it can be tested on its own:

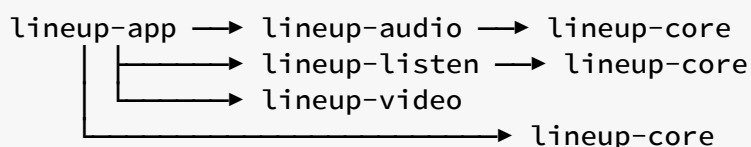
Crate	Kind	Job
<code>lineup-core</code>	library	The music and the song: notes, scales, chords and harmony, the chord sheet, patterns and the playlist, project files and diffs, undo history, MIDI files, lead sheets, the band, and the library of styles, rhythms and songs. No audio and no UI.
<code>lineup-audio</code>	library	The real-time engine and the offline renderer: drums and samplers, the Lineup Osc, SoundFont instruments, song and pattern playback, effects, WAV export, and the trace of every note played.
<code>lineup-listen</code>	library	Decodes a recording (MP3, WAV, FLAC or Ogg Vorbis) and listens for its beats, key, chords and meter. Experimental.

Crate	Kind	Job
<code>lineup-video</code>	library	Paints the app's views in memory and encodes them as H.264 and MP3 in an MP4 file.
<code>lineup-app</code>	binary (program)	The window, built with <code>egui</code> , and MIDI through <code>midir</code> . It owns the state that joins the other four, and produces the <code>ferrous-lineup</code> executable.

Why five crates instead of one?

- **Clear boundaries.** `lineup-core` *cannot* accidentally depend on the UI, because it does not list `eframe` as a dependency. The compiler enforces the architecture for us.
- **Faster tests.** `lineup-core` depends only on `serde` and `ron`, so `cargo test -p lineup-core` compiles a small crate, not the whole GUI toolkit, the audio drivers or the video encoders.
- **Heavy dependencies stay where they are needed.** Only `lineup-listen` decodes audio formats, and only `lineup-video` compiles the H.264 and MP3 encoders from their C sources.
- **Reuse.** A command-line renderer or a plugin could be built on top of `lineup-core` and `lineup-audio` without touching the window code. The playback trace already works this way: `ferrous-lineup --trace` plays a song through `lineup-audio` without opening a window (see [The playback trace](#)).

The dependencies only point one way:



`lineup-video` does not know about songs at all: it is given pictures and sound and makes a file of them.

Package name and program name

The program is built from the package `lineup-app`, but the file Cargo writes is called `ferrous-lineup`. The two names live in different places of `crates/lineup-app/Cargo.toml`:

```
[package]
name = "lineup-app"

[[bin]]
name = "ferrous-lineup"
path = "src/main.rs"
```

`[package] name` is how Cargo and other crates refer to the package (`cargo test -p lineup-app`). `[[bin]]` describes one program the package builds; the double brackets are TOML for “one entry of a list”, because a package can build several programs. Without a `[[bin]]` entry, Cargo would name the program after the package, `lineup-app` . A crate called `lineup-core` is written `lineup_core` in use lines, because a hyphen is a minus sign in Rust and cannot be part of a name; Cargo turns the hyphens into underscores for you.

The workspace

A **workspace** is a group of crates that are built together and share one `Cargo.lock` file and one `target/` output folder. Look at the top-level `Cargo.toml` :

```
[workspace]
resolver = "3"
members = [
    "crates/lineup-core",
    "crates/lineup-audio",
    "crates/lineup-listen",
    "crates/lineup-video",
    "crates/lineup-app",
]

[workspace.package]
version = "0.4.0"
edition = "2024"
license = "GPL-3.0-or-later"
repository = "https://github.com/naelolaiz/ferrous-lineup"
rust-version = "1.95"

[workspace.dependencies]
lineup-core = { path = "crates/lineup-core" }
cpal = "0.18"
...
```

- `members` lists the crates.
- `[workspace.package]` holds settings shared by every crate. Each crate then writes `version.workspace = true` to inherit them, so the version number lives

in one place. `rust-version` is the oldest Rust compiler that can build the code; an older one stops with a clear message instead of a confusing error.

- `[workspace.dependencies]` pins each dependency version once. A crate says `cpal.workspace = true` to use it. This keeps all crates on the same version of every library.

A version like `"0.18"` means "0.18.0 or any newer 0.18.x". The exact versions Cargo picked are recorded in `Cargo.lock`, which is committed to the repository so everyone builds exactly the same code.

Editions

`edition = "2024"` picks the **edition** of the Rust language. Editions let Rust make small breaking changes every few years without breaking old code: each crate declares which edition it is written in, and crates of different editions work together fine.

Modules

Inside a crate, code is organized in **modules**. A module is usually one file. In

`crates/lineup-core/src/lib.rs`:

```
pub mod pitch;    // loads src/pitch.rs as the module `pitch`
pub mod tempo;    // loads src/tempo.rs as the module `tempo`
// ... and about thirty more

pub use pitch::Note;    // re-export: users can write lineup_core::Note
pub use tempo::Tempo;
```

- `mod pitch;` tells the compiler to look for `pitch.rs` next to `lib.rs`.
- `pub` makes it visible outside the crate. Without `pub`, everything in Rust is **private** by default, visible only inside its own module.
- `use` brings a name into scope, like `import` in other languages. `pub use` also re-exports it.

A module can have modules of its own. `harmony.rs` declares `pub mod chord;`, and the compiler finds it in `harmony/chord.rs`: the folder has the name of the module it belongs to.

`lib.rs` is the **crate root** of a library; `main.rs` is the crate root of a program. Note that the crate is called `lineup-core` in `Cargo.toml` but `lineup_core` in code: Rust

code cannot contain `-` in names, so Cargo replaces it with `_`.

Doc comments

Comments starting with `///` document the item below them, and `//!` documents the enclosing module or crate. They are written in Markdown. Run `cargo doc --open` to see them rendered as a browsable website: every type and function in Ferrous Lineup shows up with its explanation.

A first type: Note

Most of `lineup-core` is small types with functions attached. The smallest, `Note` in `pitch.rs`, shows the pieces every later chapter uses. A note is stored as its MIDI number, 0 to 127, middle C being 60:

```
[derive(Debug, Clone, Copy, PartialEq, Eq, PartialOrd, Ord, Hash,
Serialize, Deserialize)]
[serde(from = "u8", into = "u8")]
pub struct Note(u8);
```

`Note` is a **tuple struct**: a struct whose fields have no names, read as `.0`, `.1` and so on. With one field that gives a number a type of its own, it is a *newtype*, which [Pitch classes, notes and spelling](#) explains with the rest of the note's music. The field has no `pub`, so code outside `pitch.rs` cannot write `Note(200)`.

Methods: `impl` blocks

A type's functions are written apart from its data, in an `impl` block:

```
impl Note {
    /// The lowest MIDI note.
    pub const MIN: Note = Note(0);
    /// The highest MIDI note.
    pub const MAX: Note = Note(127);
    /// Middle C.
    pub const C4: Note = Note(60);
    // ...

    /// Creates a note from a MIDI number, clamping it to 0..=127.
    pub fn new(midi: u8) -> Self {
        Note(midi.min(Self::MAX.0))
    }

    /// The MIDI note number.
    pub fn midi(self) -> u8 {
        self.0
    }
    // ...
}
```

- `pub const C4` is an **associated constant**, written `Note::C4`.
- `Self` is the type the `impl` is for, here `Note`.
- A function whose first parameter is `self` is a **method**, called as `note.midi()`. One without `self`, like `new`, is an **associated function**, called as `Note::new(60)`. Rust has no constructors; `new` is only the usual name.
- The last expression of a function, written without `;`, is what it returns.
- `midi` takes `self` by value, not `&self`: a `Note` is `Copy`, so passing one copies a single byte.

Traits, and derive

A **trait** is a set of methods a type promises to have, like an interface in other languages. The standard library's `fmt::Display` says how a value is written by `{}` in `format!` and by `to_string()`:

```
impl fmt::Display for Note {
    /// Writes the note as a name and octave, such as `C#4`.
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(f, "{}{}", self.pitch_class(), self.octave())
    }
}
```

So `Note::new(61).to_string()` is `"C#4"`, and every place that shows a note writes it the same way.

Many traits are the same for every type: comparing field by field, copying, printing for debugging. `#[derive(...)]` asks the compiler to write those for you:

Trait	Gives
Debug	<code>{:?}</code> printing, for debugging and test failures
Clone, Copy	a copy of the value; with <code>Copy</code> , assigning copies instead of moving
PartialEq, Eq	<code>==</code> and <code>!=</code>
PartialOrd, Ord	<code><</code> , <code>></code> and sorting
Hash	use as a key in a <code>HashMap</code>

`Serialize` and `Deserialize` come from `serde`, not from the compiler ([Saving](#)).

`Default` is the trait for “a sensible starting value”; `Tempo` writes its own, 130 BPM:

```
impl Default for Tempo {
    fn default() -> Self {
        Tempo { bpm: 130.0 }
    }
}
```

Numbers: as

Rust never changes a number’s type by itself. `as` says so, as in `octave`:

```
pub fn octave(self) -> i8 {
    (self.0 / 12) as i8 - 1
}
```

`self.0 / 12` is a `u8`, which cannot be negative, and the octave of the lowest notes is -1, so the division’s result becomes an `i8` before 1 is taken away. Indexes into a list must be `usize`, so `as usize` shows up wherever a number picks an item.

Panics

When a program reaches something that must never happen, such as an index past the end of a list, Rust **panics**: it stops the thread and prints where and why. `assert!`, `assert_eq!`, `unwrap` and `expect` panic on purpose when their condition fails, which is how a test fails ([How the code is tested](#)). A panic means a mistake in the program,

never bad input: a damaged project file is an error the app shows, and tests make sure no file can make the reader panic ([Saving](#)).

Rules the whole program keeps

A handful of decisions shape every part of the code. Each has a chapter that shows it at work:

1. **The audio thread never waits.** The sound card asks for samples every few milliseconds, and a late answer is a click. The audio thread never takes a lock, never allocates memory and never makes a system call; it reads commands from a queue and leaves its messages in others. See [Two threads](#).
2. **The window owns the project; the audio thread keeps its own copy** of just what it needs to make sound. The two never share data, so they never need a lock.
3. **One way to tell the audio thread.** Every edit, and every undo and redo, compares the project before and after and sends the differences. There is one piece of code to get right, and it is tested on its own. See [Telling the audio thread what changed](#).
4. **Keep copies, not instructions.** Undo keeps whole copies of the project, not a list of how to reverse each edit. A copy cannot be reversed wrongly. See [Undo and redo](#).
5. **Things that refer to each other by number change together.** A clip says “pattern 2”; remove pattern 1, and the clip must now say “pattern 1”. The project owns all its lists and only lets them change through its methods. See [The project](#).
6. **A file is a contract with the future.** The file format has its own types, separate from the ones the program works with, and a version number that only moves when it must. See [Saving](#).
7. **Drawing is separated from changing.** A window’s drawing code returns a list of actions, and the app applies them in one place, through the same edit function. See [The editors](#).
8. **Music theory as types.** Pitch classes, scales, chords and “what is playing” are small Rust types with methods, tested like any other code. See [Pitch classes, notes and spelling](#).
9. **Tools are plain functions.** Every editing tool takes notes and returns new ones, knowing nothing about windows or mice. See [Editing tools as plain functions](#).
10. **Knowledge as data.** Styles, rhythms and songs are written as short strings in tables that a person can read and check, in Roman numerals and steps of a bar, so the same table plays in any key and meter. See [The library](#).

11. **The band only writes patterns.** Trying a style or playing a chord sheet is an ordinary edit, so undo, saving and the audio thread work without knowing the band exists. See [How the band plays a chord sheet](#).
12. **The same code, faster than real time.** Exporting a WAV file, a video or a trace runs the very same code the sound card uses, on a thread of its own, as fast as the computer can go. See [Exporting](#).

Where each part lives

Every source file starts with a `//!` comment that says what it is for; this map is made of their first lines. `lineup-core`:

Module	What it holds
<code>pitch.rs</code> , <code>tempo.rs</code> , <code>meter.rs</code>	notes and their names, tempo, time signatures
<code>pattern.rs</code> , <code>project.rs</code> , <code>playlist.rs</code> , <code>arrangement.rs</code>	patterns, channels, tracks and clips, and the song as it plays
<code>sequencer.rs</code> , <code>swing.rs</code>	where playback is, step by step
<code>diff.rs</code> , <code>history.rs</code> , <code>file.rs</code>	what changed, undo, saving
<code>harmony/</code>	scales, chords, symbols, keys, sets, rows, micromodes, hearing harmony, voice leading
<code>tools.rs</code>	the piano roll's tools
<code>chord_sheet/</code> , <code>band/</code> , <code>bars.rs</code>	the chord sheet and the band that plays it
<code>library/</code>	styles, rhythms, songs and demos, written as data
<code>lead_sheet/</code>	the tune in notation
<code>midi.rs</code> , <code>midi_file.rs</code> , <code>record.rs</code>	MIDI messages, MIDI files, recording
<code>mixer.rs</code> , <code>effects.rs</code> , <code>score_order.rs</code>	the mixer's settings
<code>i18n/</code>	the Spanish for every text the app writes

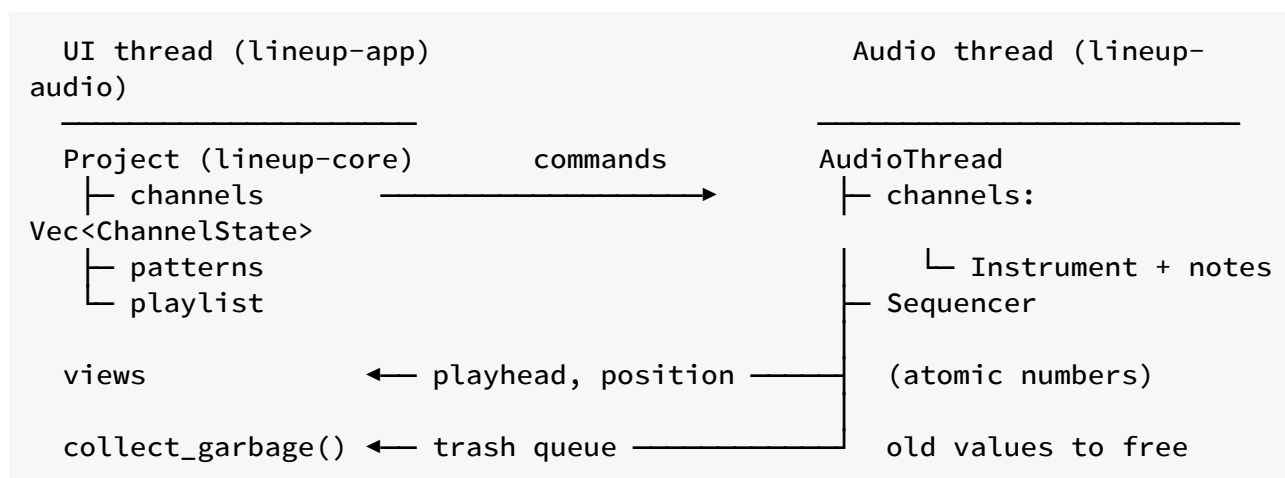
`lineup-audio`: `engine.rs` (the sound card and the audio thread), `instrument.rs`, `drums.rs`, `timbre.rs`, `sample.rs` and `soundfont.rs` (the instruments), `effects.rs` and `reverb.rs`, `levels.rs`, `metronome.rs`, `render.rs` (playing without a sound card) and `trace.rs`.

lineup-app: `app.rs` and the modules in `app/` (`menus`, `files`, `editing`, `export`, `MIDI`, `settings`), one module per window (`channel_rack.rs`, `piano_roll.rs`, `playlist.rs`, `chord_sheet.rs`, `harmony_keys.rs`, `library_browser.rs`, `lead_sheet.rs`, `twelve_tone.rs`, `micromodes.rs`, `style_editor.rs`, `ear_training.rs`), and the shared pieces: `theme.rs`, `widgets.rs`, `icons.rs` and `i18n.rs`.

Two threads: the window and the audio

Files: `crates/lineup-audio/src/engine.rs`, `crates/lineup-audio/src/dsp.rs` and `crates/lineup-app/src/midi.rs`

The window and the sound live on different threads. The window thread draws about sixty times a second and answers clicks and keys; the audio thread fills the sound card's buffer every few milliseconds and must never be late. How the two talk to each other is the most important decision in the program, and it is where Rust's ownership rules start to earn their keep.



How digital audio works

Sound is a vibration. A sound card plays it by moving a speaker according to a list of numbers called **samples**, typically 44,100 or 48,000 per second (the **sample rate**). Each sample is a value between -1.0 and 1.0, the speaker position at that instant. With two **channels** (stereo), the samples are **interleaved**: left, right, left, right...

The sound card does not wait for us. It plays from a small buffer and, every few milliseconds, asks the program to refill it by calling a function we provide: the **audio callback**. It runs on a separate, high-priority **audio thread**.

The golden rule of the audio thread

If the callback is late, even once, the buffer runs empty and you hear a click or a gap (an *underrun* or *xrun*). So code on the audio thread must never do anything that can

take an unpredictable amount of time:

- no **locks** (a `Mutex` held by the UI could block it),
- no **memory allocation or freeing** (`Vec::push` past its capacity, `String`, `Box::new`, or dropping the last owner of any of them),
- no **file, network or console I/O** (`println!` included),
- no waiting for other threads.

Everything else in this chapter follows from this rule.

Sending commands with a ring buffer

The UI never changes audio state directly. It sends messages, values of one enum with nearly forty variants. A few of them:

```
pub enum Command {
    Play,
    Seek(usize),
    Stop,
    SetTempo(Tempo),
    SetMasterVolume(f32),
    InsertChannel {
        index: usize,
        instrument: Instrument,
        notes: ChannelNotes,
        volume: f32,
        muted: bool,
    },
    RemoveChannel(usize),
    SetNotes {
        pattern: usize,
        channel: usize,
        notes: Vec<NoteEvent>,
    },
    // ...
}
```

This is a Rust **enum**. Unlike enums in C or Java, each variant can carry its own data: `Seek` carries a step number, `SetNotes` a whole list of notes. An enum value is exactly one of its variants.

The messages travel through a **ring buffer** from the `rtrb` crate: a fixed-size queue, allocated once, with one writer (the `Producer`) and one reader (the `Consumer`). Pushing and popping never lock and never allocate, so the audio thread can read it safely.

```
let (command_producer, command_consumer) =
    rtrb::RingBuffer::new(COMMAND_QUEUE_SIZE);
let (trash_producer, trash_consumer) =
    rtrb::RingBuffer::new(TRASH_QUEUE_SIZE);
let (live_producer, live_consumer) =
    rtrb::RingBuffer::new(LIVE_QUEUE_SIZE);
let (midi_producer, midi_consumer) =
    rtrb::RingBuffer::new(MIDI_OUT_QUEUE_SIZE);
```

`RingBuffer::new` returns a **tuple** of two values, which we take apart with a `let (a, b) = ...` **pattern**. There are four queues: commands from the UI, values for the UI to free (the trash, below), keys from a MIDI keyboard ([MIDI](#)), and messages for MIDI Out channels. For each, one end stays on one thread and the other end is moved to the other.

Ownership makes this safe

Here is where Rust shines. `Producer` can be used from one thread and `Consumer` from another, but there is only one of each, and Rust's **ownership** rules guarantee it:

- Every value has exactly one **owner**. When the owner goes away, the value is dropped (freed).
- Assigning or passing a value **moves** it to the new owner, and the old name can no longer be used.

Once `consumer` is moved into the audio callback, the UI code physically cannot touch it anymore: the compiler would reject it. Data races are not “unlikely”; they are compile errors.

Commands that carry data

`Command` is not `Copy`, because an `Instrument` can hold an `Arc<SampleData>` and `SetNotes` holds a `Vec`. Moving it through the ring buffer is still fine: the value is moved into the queue's preallocated slot and out again, with no allocation. The `Vec` inside was allocated on the UI thread, before it was sent.

When the queue is full

The queue holds 4096 commands, enough for a whole large project. If it is ever full, the UI got ahead of the audio thread, which empties it at the start of every block.

`send_wait` does not drop the command: it yields and tries again, up to a thousand times.

```
pub fn send_wait(&mut self, mut command: Command) -> bool {
    for _ in 0..SEND_WAIT_ROUNDS {
        match self.commands.push(command) {
            Ok(()) => return true,
            Err(rtrb::PushError::Full(returned)) => command = returned,
        }
        // A stream that has reported an error will not empty the
        // queue, so waiting for room would only freeze the window.
        if self.problems.broken.load(Ordering::Relaxed) {
            return false;
        }
        std::thread::yield_now();
    }
    false
}
```

`push` takes the command by value. When the queue is full it gives the command back inside the error, `PushError::Full(returned)`, so the loop can try again with it: nothing is copied and nothing is lost.

Opening the sound card

`AudioEngine::start` uses the `cpal` crate, which works on Linux (ALSA), macOS (CoreAudio) and Windows (WASAPI), so the same code runs on all three.

```
pub fn start() -> Result<Self, EngineError> {
    let host = cpal::default_host();
    let device = host
        .default_output_device()
        .ok_or(EngineError::NoOutputDevice)?;
    let supported = device
        .default_output_config()
        .map_err(EngineError::Config)?;
    ...
}
```

Option, Result and the ? operator

Rust has no `null` and no exceptions. Instead:

- `Option<T>` is either `Some(value)` or `None`. `default_output_device()` returns `Option<Device>` because there may be no sound card.
- `Result<T, E>` is either `Ok(value)` or `Err(error)`. `default_output_config()` returns `Result<SupportedStreamConfig, cpal::Error>`.

Both are ordinary enums from the standard library. Because they are types, you cannot forget to handle the missing or failed case: the compiler will not let you use a `Device` you might not have.

- `.ok_or(EngineError::NoOutputDevice)` turns an `Option` into a `Result`, using our error for the `None` case.
- `.map_err(EngineError::Config)` wraps `cpal`'s error in our own variant. Notice that `EngineError::Config` is used like a function here: enum variants with data can be called as constructors.
- `?` means "if this is an `Err`, return it from the function right now; otherwise give me the `Ok` value". It replaces long chains of `if error then return`.

A custom error type

```
#[derive(Debug)]
pub enum EngineError {
    NoOutputDevice,
    Config(cpal::Error),
    UnsupportedFormat(SampleFormat),
    BuildStream(cpal::Error),
    Play(cpal::Error),
}

impl fmt::Display for EngineError { ... }
impl std::error::Error for EngineError {}
```

One enum lists every way starting the engine can fail. Implementing `Display` gives each a human-readable message, and the empty `impl std::error::Error` marks it as a proper error type that works with the rest of the Rust ecosystem. The app turns it into text with `err.to_string()` and keeps running without sound: every window works, and **Edit > Settings... > Audio** offers to try again.

Generics: one function, many sample types

Different sound cards want samples in different formats: 32-bit floats, 16-bit integers, and so on. We generate `f32` internally and convert at the last moment. Instead of

writing the stream code six times, we write it once as a **generic function**:

```
fn build_stream<T>(  
    device: &cpal::Device,  
    config: StreamConfig,  
    mut audio_thread: AudioThread,  
    problems: Arc<StreamProblems>,  
) -> Result<cpal::Stream, EngineError>  
where  
    T: SizedSample + FromSample<f32>,  
    { ... }
```

`T` is a placeholder type, and the `where` clause lists **trait bounds**: `T` can be any type that is a cpal sample (`SizedSample`) and that can be created from an `f32` (`FromSample<f32>`). Inside, `T::from_sample(x)` does the conversion.

Then `start` picks the right version with a `match`:

```
let stream = match sample_format {  
    SampleFormat::F32 => {  
        build_stream::<f32>(&device, config, audio_thread,  
Arc::clone(&problems))  
    }  
    SampleFormat::I16 => {  
        build_stream::<i16>(&device, config, audio_thread,  
Arc::clone(&problems))  
    }  
    ...  
    other => return Err(EngineError::UnsupportedFormat(other)),  
};
```

- `match` compares a value against **patterns** and runs the first arm that fits. The compiler checks that every possible value is covered. `other` is a catch-all pattern that also gives a name to the value.
- `build_stream::<f32>` (the “turbofish” syntax `::<>`) chooses `T = f32` . The compiler generates a separate, fully optimized copy of the function for each type used. This is called **monomorphization**, and it means generics cost nothing at run time.
- `audio_thread` is moved into whichever arm runs. The compiler knows only one arm runs, so this is allowed.

The audio callback is a closure

Inside `build_stream`:

```

let data_callback = move |output: &mut [T], _: &cpal::OutputCallbackInfo|
{
    audio_thread.process(output);
};

let error_callback = move |err: cpal::Error| {
    problems.broken.store(true, Ordering::Relaxed);
    if let Ok(mut slot) = problems.error.lock() {
        *slot = Some(err.to_string());
    }
    eprintln!("audio stream error: {err}");
};

device
    .build_output_stream(config, data_callback, error_callback, None)
    .map_err(EngineError::BuildStream)

```

`|args| body` is a **closure**: an anonymous function that can use variables from the place where it was written. The `move` keyword makes the closure take **ownership** of `audio_thread`, and with it of the command consumer and everything else the audio thread works with. That is necessary because the closure will run on another thread, long after `build_stream` has returned, so it cannot just borrow local variables.

cpal's signature requires the callback to be `FnMut(&mut [T], &OutputCallbackInfo) + Send + 'static`:

Bound	Meaning
<code>FnMut</code>	a closure that may change the values it owns (the instruments advance their state)
<code>Send</code>	safe to move to another thread
<code>'static</code>	owns everything it uses, borrowing nothing that could disappear

If we tried to share something unsafe, such as a plain reference into the UI's data, the program would not compile. This is how Rust turns the golden rule of audio into checks the compiler does for you.

`&mut [T]` is a **mutable slice**: a borrowed view into the sound card's buffer, which we fill but do not own.

The error callback is the one place that may lock and print: cpal calls it outside the audio callback, when something goes wrong while the stream runs, such as an unplugged interface. It keeps the newest problem for the window to show.

AudioThread::process

The audio callback hands each buffer to one method. Its outline:

```
pub(crate) fn process<T>(&mut self, output: &mut [T])
where
    T: SizedSample + FromSample<f32>,
{
    self.flush_trash();
    self.flush_midi_backlog();
    while let Ok(command) = self.commands.pop() {
        self.apply(command);
    }
    while let Some(message) = self.live_input.as_mut().and_then(|keys|
keys.pop().ok()) {
        self.play_live(message);
    }

    for frame in output.chunks_mut(self.output_channels) {
        if self.playing
            && let Some(step) = self.sequencer.tick()
        {
            self.playhead.store(step, Ordering::Relaxed);
            self.start_notes(step);
        }
        // ... mix the channels, the reverb and the metronome ...
    }
    self.send_midi();
}
```

- `while let Ok(command) = ...` keeps looping as long as the pattern matches, so it drains every waiting command and stops when the queue is empty.
- `as_mut` turns `&mut Option<Consumer>` into `Option<&mut Consumer>`, and `and_then` asks it for a key only if there is a consumer.
- `chunks_mut(channels)` splits the buffer into **frames**, one sample per output channel.
- `if a && let Some(x) = b { ... }` is a **let chain**: a condition and a pattern match combined in one `if`.

The audio thread's state lives in a struct, `AudioThread`, instead of loose variables inside the closure. That lets the tests run it with a plain `Vec` instead of a sound card, and lets the `renderer` play a song into a file with the same code.

Mixing a frame

For each frame, every channel's instrument makes one sample. The sample goes through the channel's effects and volume, and then to the left and right sides as its pan says, and to the reverb as much as its send says ([The mixer](#)):

```
for channel in &mut self.channels {
    let volume = channel.volume.next_value();
    let sample = channel.effects.process(channel.instrument.next_sample())
* volume;
    ...
    dry[0] += kept * gains[0];
    dry[1] += kept * gains[1];
    send += sample * amount;
}
...
// `tanh` gently rounds off peaks above 1.0 instead of clipping them hard.
let stereo = stereo.map(f32::tanh);
```

- `for channel in &mut self.channels` iterates over mutable references, so each instrument can advance its own state.
- Mixing is just adding: sounds in the air add up, and so do samples.
- Several loud drums at once can add up past 1.0, which a sound card would clip harshly. `tanh` (the hyperbolic tangent) passes small values almost unchanged and bends large ones smoothly toward ± 1.0 , a gentle **soft clipper**.
- `volume.next_value()` is a `Smoothed` value. A volume that jumps makes the speaker jump, which you hear as a click, so it glides toward its target instead: every sample it covers a small fraction of the remaining distance. This is a *one-pole smoother*, settling in a few milliseconds, too fast to hear as a fade but enough to remove the click.

Room reserved in advance

```
channels: Vec::with_capacity(MAX_CHANNELS),
```

A `Vec` has a **length** (how many items it holds) and a **capacity** (how many fit in the memory it has). `push` allocates only when the length would exceed the capacity. Reserving room for `MAX_CHANNELS` at startup, on the UI thread, means `push` on the audio thread never allocates. The UI refuses to add more channels than that, so the limit is never hit. The list of notes waiting to start late, the pattern lengths and the trash backlog are reserved the same way.

`Vec::remove(index)` shifts the following items down, which never allocates either. That is how `RemoveChannel` keeps the audio thread's channel numbers matching the UI's list.

Freeing memory somewhere else: the trash queue

When a channel is removed or its notes replaced, the old value may hold the last `Arc` to a sample, or a `Vec` of notes. Dropping it would free that memory, and freeing can take a lock inside the memory allocator. So the audio thread does not drop it. It sends it back:

```
fn set_instrument(&mut self, channel: usize, instrument: Instrument) {
    match self.channels.get_mut(channel) {
        Some(state) => {
            let mut old = std::mem::replace(&mut state.instrument,
instrument);
            silence(&mut old, &mut self.midi_out, &mut self.midi_backlog);
            ...
            self.throw_away(Garbage::Instrument(old));
        }
        None => self.throw_away(Garbage::Instrument(instrument)),
    }
}
```

`std::mem::replace(&mut place, new)` puts `new` into `place` and returns what was there. It is how you take a value out of a struct field you only have a `&mut` to, since Rust never lets a field be left empty. (`silence` sends the note-offs the old instrument still owes a MIDI Out synth.)

`throw_away` pushes the value into a second ring buffer, going the other way. `Garbage` is an enum with a variant for each kind of thing the audio thread gives back: instruments, note lists, channels, songs, tempo lists, phrases, delay lines. The UI thread empties the queue once per frame:

```
pub fn collect_garbage(&mut self) {
    while let Ok(garbage) = self.trash.pop() {
        // Dropping a value frees its memory. Here, on the UI thread,
        // that is allowed to take as long as it needs.
        match garbage {
            Garbage::Instrument(instrument) => drop(instrument),
            Garbage::Notes(notes) => drop(notes),
            ...
        }
    }
}
```

`drop(value)` is a plain function from the prelude that takes ownership and lets the value go out of scope. Writing it explicitly documents the intent: this is where the memory is freed.

A burst of changes between two frames could fill the trash queue. Then `throw_away` keeps the value in a backlog reserved in advance, and `flush_trash` moves it into the queue at the start of the next block. Only if the backlog fills too, which means the UI has not collected for a long time, is the value freed on the audio thread, as a last resort.

The playhead: an atomic integer

The UI wants to highlight the step being played. The audio thread knows it. Sharing it through a `Mutex` would mean the audio thread could wait for the lock, so we use an **atomic**:

```
let playhead = Arc::new(AtomicUsize::new(NO_STEP));
// audio thread:
self.playhead.store(step, Ordering::Relaxed);
// UI thread:
match self.playhead.load(Ordering::Relaxed) { NO_STEP => None, step =>
Some(step) }
```

An `AtomicUsize` is a number that the processor can read and write in one indivisible operation, so two threads can use it without a lock and without ever seeing a half-written value. Both threads hold an `Arc` to the same one (`Arc::clone(&playhead)` makes the second handle).

`Ordering::Relaxed` says we only care about this one value, not about the order of other memory writes around it. It is the cheapest ordering and enough for a display. Stricter orderings (`Acquire`, `Release`, `SeqCst`) are for when one value guards access to others.

Rust has no “null” step, so `usize::MAX` stands for “stopped”, and `playhead()` turns it into an honest `Option<usize>` right away so no other code ever sees the magic number.

The position to a fraction of a step, which recording and the views need, is an `f64`, and there is no atomic float. So it is kept as the bits of the `f64` in an `AtomicU64`: `position.to_bits()` on the audio thread, `f64::from_bits(bits)` on the other side. Two numbers that must always be read together, such as the pattern played over the

song and its step, share one `AtomicU64`: one in the high 32 bits, the other in the low ones.

Keeping the stream alive: Drop

```
pub struct AudioEngine {  
    // The stream is never read, but it must be kept: dropping it stops  
    audio.  
    _stream: cpal::Stream,  
    commands: rtrb::Producer<Command>,  
    trash: rtrb::Consumer<Garbage>,  
    playhead: Arc<AtomicUsize>,  
    position: Arc<AtomicU64>,  
    ...  
}
```

The `cpal::Stream` is never used after it starts, but it must be kept: when a value is dropped, Rust runs its cleanup code (its `Drop` implementation), and dropping a `Stream` closes the sound card. Storing it in the engine ties the sound's lifetime to the engine's. The leading underscore tells the compiler "I know this field is never read".

This pattern, where owning a value means owning a resource and dropping it releases the resource, is called **RAII**. Rust uses it everywhere: files close, locks unlock and memory frees itself when its owner goes away.

When the sound card goes away

An unplugged interface, a changed default device or a sound server that restarted makes the stream report an error. **Edit > Settings... > Audio** then offers **Try again** (and **Restart audio** while the stream runs), which opens the system's default output device again. The project stays: the whole of it is sent to the fresh audio thread, the MIDI ports are reconnected to the new engine, and a song that was playing seeks back to the same position and plays on.

The old engine is dropped before the new one starts, because a device being reopened cannot still be held by the dead stream.

Two things need care when an audio thread goes away:

Notes sounding on an external synth. A note sent over MIDI Out gets its note-off from the audio thread that played it; when that thread is gone, the note-offs it still

owed go with it. The messages go out through a small thread, `midi-out`, that empties the audio thread's queue every millisecond. When its queue is abandoned, its loop ends, and before it finishes it sends **All Notes Off** (controller 123) on all sixteen channels:

```
let sender = move || {
    while !messages.is_abandoned() {
        // ... send what the audio thread queued ...
    }
    // The engine is gone, and with it the note offs of notes still
    // sounding on the synth.
    if let Some(connection) = lock(&output).as_mut() {
        for channel in 0..16 {
            let _ = connection.send(&all_notes_off(channel));
        }
    }
};
```

The new engine's sending thread must not start before the old one has sent its All Notes Off, or that message could cut off the new engine's first notes. So `reconnect_audio` waits for the old thread, polling `JoinHandle::is_finished` for at most 100 ms (in practice about a millisecond, since the old engine is already gone). It does not call `join()`, which could wait for ever on a MIDI port that hangs: the window must never freeze over a port.

Messages that must arrive. A dropped note-off leaves a note sounding for ever on an external synth, so MIDI Out never drops one. Messages that find the output queue full wait in a reserved backlog and go out at the start of the next block. Removing a channel or replacing an instrument sends its note-offs the same way, and disconnecting a port sends All Notes Off on all sixteen channels before the connection closes.

The [playback trace](#) is the opposite case: it records every note for a report, not for the sound, so if its queue fills, the audio thread counts the lost events and carries on. Waiting there would put the trace before the music. When the audio restarts, the trace goes on in the same file: what the old thread left is written, and a new queue is sent to the new thread.

Testing without a sound card

The tests at the bottom of `engine.rs` build an `AudioThread` with ring buffers, push commands, render into a `Vec<f32>` and check the result: a hit at frame 0 and another 6000 frames later at 120 BPM, silence when muted, edits taking effect while playing,

and old instruments arriving in the trash. A small `Harness` struct holds the pieces so each test stays short.

The project: patterns, channels, tracks, clips and markers

Files: `crates/lineup-core/src/pattern.rs`, `crates/lineup-core/src/project.rs`, `crates/lineup-core/src/playlist.rs` and `crates/lineup-core/src/arrangement.rs`

Everything the user works on is one value of type `Project`: the channels of the channel rack, the patterns of notes they play, the playlist that arranges the patterns into a song, and the markers along it (sections, tempo, time signature, key and chords). It is also exactly what a project file saves. This chapter is about its shape and the rules it keeps. The windows that edit it are in [The editors](#); how to use them is in the user manual's [Patterns and the playlist](#).

Notes, steps and patterns

A **pattern** is a short stretch of music, often one bar, with a row of notes for every channel. A note is a pitch, a start, a length and a velocity, all counted in **steps** (a sixteenth note each in 4/4):

```
pub struct NoteEvent {
    pub pitch: Note,
    /// The step the note starts on, counted from 0.
    pub start: usize,
    /// How many steps the note is held for, at least 1.
    pub length: usize,
    /// How hard the note is played, from 1 to 127 (as in MIDI).
    pub velocity: u8,
    /// How many ticks after the start of its first step the note really
    /// starts, from 0 to [`TICKS_PER_STEP`] - 1.
    pub nudge: u8,
    /// How many ticks before the end of its last step the note really
    /// ends.
    pub trim: u8,
}
```

The channel rack's step buttons and the piano roll edit the same notes. A step that is "on" is just a note one step long at the default pitch, so `is_on` asks whether any note starts there:


```
pub fn is_on(&self, channel: usize, step: usize) -> bool {
    self.notes(channel).iter().any(|note| note.start == step)
}
```

One kind of data with two views is how pattern-based studios work, and it means the audio thread has a single kind of thing to play. `nudge` and `trim` place a note between steps, for music played by a person or imported from a MIDI file; notes on the grid leave them at 0.

Channels: enums with data

```
pub enum SampleSource {
    /// One of the drums built into Ferrous Lineup.
    Builtin(Drum),
    /// A WAV file on disk.
    File(PathBuf),
}

pub enum ChannelKind {
    /// Plays a recorded or built-in sound from start to end for each
    /// note, faster for higher notes and slower for lower ones.
    Sampler(SampleSource),
    /// The built-in "Lineup Osc" synthesizer, which can play chords.
    Synth,
    /// Makes no sound itself: sends its notes out through MIDI, on
    /// MIDI channel `channel` (0 to 15, shown as 1 to 16).
    MidiOut { channel: u8 },
}
```

A channel has a name, a volume, a mute switch, a General MIDI instrument number, and a kind. Enums nest: a channel is a sampler, the Lineup Osc or a MIDI Out channel, and a sampler's sound is either a built-in drum or a file. Each fact is stored once and every combination makes sense. `MidiOut { channel: u8 }` shows that a variant can also have **named fields**, like a struct.

`PathBuf` is the standard type for an owned file path. It is not a `String`, because file names on some systems are not valid UTF-8 text; `path.display()` turns one into something printable.

`Drum::ALL` is an associated constant holding every variant, so menus can loop over the drums instead of listing them by hand.

The shape of a project

A project has a list of patterns, and a playlist that says where each one plays:

```
pub struct Project {
    pub mixer: crate::Mixer,
    pub tempo: Tempo,
    pub scale: Scale,
    pub time_signature: TimeSignature,
    pub loose_timing: u8,
    pub swing: u8,
    pub(crate) channels: Vec<Channel>,
    pub(crate) patterns: Vec<Pattern>,
    pub(crate) playlist: Playlist,
    pub(crate) band: Option<Band>,
    // ...where each part came from, and the locked channels
}
```

Every pattern has one row of notes per channel. Pattern-based studios work this way: a pattern is “what every instrument plays during these bars”, so the drums pattern simply has empty rows for the synth.

The playlist is two lists:

```
pub struct Track {
    pub name: String,
    pub muted: bool,
    // ...a display height, which older files leave out
}

pub struct Clip {
    pub pattern: usize,
    pub track: usize,
    pub start: usize,
    /// How many times the pattern plays, one after the other.
    pub repeats: usize,
}

pub struct Playlist {
    pub(crate) tracks: Vec<Track>,
    pub(crate) clips: Vec<Clip>,
    // ...sections, a loop region, and time, tempo, key and chord markers
}
```

A clip is three numbers and a repeat count: which pattern, on which track, starting at which step of the song, and how many times the pattern plays. Notice what is *not* there: the clip’s length. It is the pattern’s length times the number of repeats, so it is worked out when needed:

```
pub fn clip_length(&self, clip: &Clip) -> usize {
    self.patterns.get(clip.pattern).map_or(0, Pattern::length) *
    clip.repeats
}

impl Clip {
    /// A clip that plays its pattern once.
    pub fn new(pattern: usize, track: usize, start: usize) -> Self {
        Clip { pattern, track, start, repeats: 1 }
    }
}
```

Storing the length too would mean two places that must agree. Make the pattern longer and every clip's stored length would be wrong. A value that can be computed from others is usually better computed than stored. The same goes for the end of a repeated clip: it is `start + clip_length`, not another field.

`Project::playback_arrangement` unrolls the repeats into `Play`s before the song reaches the audio thread (see [Sections, repeats and loops](#) below); the clip is the thing you draw, and its `repeats` is the little "x2" on it.

`map_or(0, Pattern::length)` says "if there is a pattern, call `Pattern::length` on it; otherwise use 0". A method can be passed by name, like a function, when its only argument is `self`.

Four rules, one gatekeeper

The comment on `Project` lists the rules it keeps:

- every clip in the playlist refers to a pattern and a track that exist;
- sections are sorted and never overlap, and repeats stay in range.

Most of these rules are about two lists at once, so no single list can keep them. The `Project` owns all of them, keeps the fields private to `lineup-core`, and changes them only in methods that keep every rule. Adding a channel adds a row to every pattern:

```
pub fn add_channel(&mut self, channel: Channel) -> Option<usize> {
    if self.channels.len() >= MAX_CHANNELS {
        return None;
    }
    self.channels.push(channel);
    for pattern in &mut self.patterns {
        pattern.push_row();
    }
    Some(self.channels.len() - 1)
}
```

Removing a pattern is the interesting one. Clips refer to patterns by their position in the list, so when pattern 1 goes, the clips of pattern 1 must go too, and the clips of patterns 2, 3, 4... must each move down by one so they keep playing the same music:

```
pub fn remove_pattern(&mut self, index: usize) -> Option<Pattern> {
    if index >= self.patterns.len() || self.patterns.len() == 1 {
        return None;
    }
    self.playlist.clips.retain(|clip| clip.pattern != index);
    for clip in &mut self.playlist.clips {
        if clip.pattern > index {
            clip.pattern -= 1;
        }
    }
    Some(self.patterns.remove(index))
}
```

- `retain` keeps the items for which the closure returns `true`, and drops the others, in one pass.
- `for clip in &mut self.playlist.clips` gives a mutable reference to each clip in turn, so `clip.pattern -= 1` changes the clip in the list.
- The method returns the removed pattern, wrapped in `Some`. The caller can ignore it, keep it, or check that the removal happened at all.

The test `removing_a_pattern_renumbers_its_followers` checks the whole story: after removing “Beat”, the five Beat clips are gone, “Chords” is pattern 0, and the fill still plays at bar 8.

For a list of at most 999 patterns (usually far fewer), positions plus one careful method are the simpler choice. It is the kind of trade-off with no single right answer; one of the [exercises](#) of this part asks you to think about it.

Handing out &mut Pattern safely

The app needs to change notes in a pattern. `Project` could copy every `Pattern` method (`toggle`, `add_note`, `replace_note` ...) as a method of its own. Instead it hands out the pattern itself:

```
pub fn pattern_mut(&mut self, index: usize) -> Option<&mut Pattern> {
    self.patterns.get_mut(index)
}
```

Is that safe? With `&mut Pattern`, could the app add a row and break the “one row per channel” rule? No, because of how the row methods are declared in `pattern.rs`:

```
pub(crate) fn push_row(&mut self) { ... }
pub(crate) fn remove_row(&mut self, channel: usize) { ... }
```

`pub(crate)` means “public inside this crate only”. `project.rs` can call them, because it is in `lineup-core`; `lineup-app` cannot, even with a `&mut Pattern` in its hands. Every method `lineup-app` can call keeps the row count as it is. Privacy in Rust is about modules and crates, not about who holds a reference, and that lets one type give out a reference to another without giving away its rules.

On the app side, every pattern edit goes through one small helper:

```
fn edit_pattern(&mut self, label: &'static str, change: impl FnOnce(&mut
Pattern)) {
    let index = self.pattern;
    self.edit(label, |project| {
        if let Some(pattern) = project.pattern_mut(index) {
            change(pattern);
        }
    });
}

// For example:
RackAction::ToggleStep { channel, step } => {
    self.edit_pattern("change steps", |pattern| {
        pattern.toggle(channel, step);
    });
}
```

- `impl FnOnce(&mut Pattern)` accepts any closure that takes a pattern. `FnOnce` is the most accepting of the three closure traits: the closure only has to be callable once, so it may even move values it captured.
- `self.edit(label, ...)` is the one function every change to the project goes through. It keeps the change for undo and tells the audio thread what changed

(Undo and redo, Telling the audio thread what changed). `label` is what **Edit > Undo** will say.

Names, and `impl Into<String>`

Patterns have names. The constructor takes the name first:

```
pub fn new(name: impl Into<String>, channels: usize, length: usize) ->
Self {
    Pattern {
        name: name.into(),
        color: 0,
        time: None,
        player: None,
        length: length.clamp(1, MAX_STEPS),
        rows: vec![Vec::new(); channels],
    }
}
```

`impl Into<String>` accepts anything that can be turned into a `String`: `Pattern::new("Beat", 5, 16)` with a `&str`, or `Pattern::new(format!("Pattern {n}"), 5, 16)` with a `String`, without the caller writing `.to_string()`. Inside, `name.into()` does the conversion (for a `String`, it just moves it).

`name` is a public field while `length` and `rows` are private. Mixing is fine in Rust: `name` has no rule to keep, so there is no reason to hide it behind methods. `color`, `time` and `player` say which colour the pattern is drawn in, the time signature it was written in, and which of the band's players wrote it, if one did.

Cloning a pattern uses `Pattern`'s `Clone` derive:

```
pub fn clone_pattern(&mut self, index: usize) -> Option<usize> {
    let mut copy = self.patterns.get(index)?.clone();
    copy.name = format!("{}", copy);
    // A copy is yours, not the band's.
    copy.player = None;
    self.push_pattern(copy)
}
```

`.clone()` copies the whole pattern, every row and every note, so the copy and the original never share anything; the test `cloned_patterns_are_independent` checks that changing one leaves the other alone. The `?` after `get(index)` returns `None` from the function straight away if there is no such pattern.

No two notes in the same place

`add_note` refuses to add a note in the same place as another. `replace_note`, used when dragging, must refuse too, or one note could be dragged exactly on top of another, so it checks every note except the one being replaced:

```
let duplicate = row
    .iter()
    .enumerate()
    .any(|(other_index, other)| other_index != index &&
other.same_place(&note));
```

So “no two notes of the same pitch start at the same moment” is true for every pattern, and loading a file can insist on it. `same_place` compares the pitch, the step and the nudge, so two notes that start a few ticks apart are still two notes.

How long is the song?

The song ends where its last clip ends, rounded up to a whole bar:

```
pub fn song_length(&self) -> usize {
    let clips = self
        .playlist
        .clips
        .iter()
        .map(|clip| clip.start + self.clip_length(clip));
    let sections = self.playlist.sections.iter().map(Section::end);
    let end = clips.chain(sections).max().unwrap_or(0);
    self.meter().round_up_to_bar(end)
}
```

`max` returns an `Option`, because an empty list has no maximum; `unwrap_or(0)` turns “no clips” into a length of 0. `chain` joins the two iterators, because a song can end with a clip or with a section that has no clip at its very end. `self.meter()` knows the time signature, and its `round_up_to_bar` is division that rounds up: in 4/4, 17 steps is `17.div_ceil(16) = 2` bars.

Where is a pattern, while the song plays?

In song mode the audio thread reports a step of the *song*. The channel rack and the piano roll show one pattern, so they need to know which step of *that* pattern is

playing, if any:

```
pub fn pattern_step_in_song(&self, pattern: usize, song_step: usize) ->
Option<usize> {
    self.playlist
        .audible_clips()
        .filter(|clip| clip.pattern == pattern)
        .find_map(|clip| self.step_in_clip(&clip, song_step))
}

/// The step of its pattern the clip plays on song step `step`.
fn step_in_clip(&self, clip: &Clip, step: usize) -> Option<usize> {
    let pattern_length = self.patterns.get(clip.pattern).map_or(0,
Pattern::length);
    let end = clip.start + pattern_length * clip.repeats;
    (clip.start..end)
        .contains(&step)
        .then(|| (step - clip.start) % pattern_length)
}
```

Read it as a sentence: of the clips that can be heard, take those of this pattern, find the first one that covers this song step, and tell me how far into the clip that step is. The `% pattern_length` is what makes a repeated clip start its pattern again each time round: a 16-step clip that repeats at song step 16 is on pattern step 0 there.

`audible_clips` skips clips on muted tracks:

```
pub fn audible_clips(&self) -> impl Iterator<Item = Clip> + '_ {
    self.clips
        .iter()
        .copied()
        .filter(|clip| self.tracks.get(clip.track).is_some_and(|track|
!track.muted))
}
```

`.copied()` turns an iterator of `&Clip` into one of `Clip`. That works because `Clip` is `Copy` (three numbers), and it means callers get plain values instead of references into the playlist.

The starting song

`Project::demo` builds an eight-bar song, which many tests play, with the same methods the app uses:


```

for bar in [0, 2, 4, 6] {
    project.add_clip(Clip::new(chords, chord_track, bar * 16));
}
for bar in 2..7 {
    project.add_clip(Clip::new(beat, drum_track, bar * 16));
}
project.add_clip(Clip::new(fill, drum_track, 7 * 16));

```

`2..7` is a range from 2 up to, but not including, 7: bars 3 to 7 when counting from 1, as the ruler does. The fill takes the place of the beat in bar 8. **File > New beat** starts from `Project::template` instead: the drum kit and a Lineup Osc, with no notes and one track. **File > New** starts from `Project::with_band`, an empty chord sheet with a band ([How the band plays a chord sheet](#)).

Sections, repeats and loops

A song has more shape than its clips: an intro, a verse, a chorus heard twice, a beat that loops for eight bars. The playlist keeps three more things for it:

- **Sections:** named stretches of the song (Intro, Verse, Chorus...) in a strip above the bar numbers. A section can be played several times in a row: it is drawn once and marked “×2”.
- **Clip repeats:** a clip can play its pattern again and again, like stretching a pattern clip in other pattern-based studios.
- **A loop region:** song mode can loop just one part of the song.

Sections do not own clips. A section is only a name and a range of steps: whatever clips lie under it are part of it. Deleting a section leaves the clips alone.

Two kinds of step

Here is the one idea that makes everything else simple. Once a section can repeat, the playlist as *drawn* and the song as *played* are no longer the same length:

```

timeline:   | Intro | Chorus ×2 | Outro |
song:       | Intro | Chorus | Chorus | Outro |

```

So the code talks about two kinds of step:

- a **timeline step** is a place in the playlist, as drawn;
- a **song step** is a moment in the song, as heard.

`Project::song_length` is the length of the timeline. `Arrangement` is the song as heard:

```
pub struct Arrangement {
    /// The pieces of patterns to play, in song steps.
    pub plays: Vec<Play>,
    /// The stretches of the timeline, in the order they play.
    pub passes: Vec<Pass>,
    /// How long the song is, in song steps.
    pub length: usize,
}

pub struct Play {
    pub pattern: usize,
    pub start: usize, // the song step it starts on
    pub skip: usize,  // how far into the clip it starts
    pub length: usize, // how many steps it lasts
}
```

A `Pass` says “timeline steps 32 to 63 play at song steps 64 to 95, and this is round 2 of 2 of section 1”. A `Play` says “play pattern 3 from song step 64 for 32 steps, starting 0 steps into it”. The pattern loops for as long as the `Play` lasts, so a clip repeated three times is one `Play` three patterns long. `skip` is not zero when a section or the loop region starts halfway through a clip.

`Project::arrangement()` builds it for the whole song (the exports use it), and `Project::playback_arrangement()` for what song mode plays: the loop region if there is one, the whole song if not.

Building the arrangement

`arrange` in `arrangement.rs` cuts the timeline into pieces at section edges. Sections play `repeats` times; the gaps between them play once:

```

fn pieces(project: &Project, from: usize, to: usize)
    -> Vec<(usize, usize, Option<usize>, usize)>
{
    let mut pieces = Vec::new();
    let mut cursor = from;
    for (index, section) in
project.playlist().sections().iter().enumerate() {
        if section.end() <= cursor || section.start >= to {
            continue;
        }
        if section.start > cursor {
            pieces.push((cursor, section.start, None, 1)); // a gap
        }
        let start = section.start.max(cursor);
        let end = section.end().min(to);
        pieces.push((start, end, Some(index), section.repeats));
        cursor = end;
    }
    if cursor < to {
        pieces.push((cursor, to, None, 1));
    }
    pieces
}

```

This only works because sections are kept **sorted and never overlapping**. That is a rule, so, as with clips, the list is private and only `Project`'s methods change it. `add_section` and `set_section` check the rule and refuse a section that would break it; `insert_section` keeps the order with `partition_point`, a binary search that returns the first place where the condition stops being true:

```

let index = sections.partition_point(|other| other.start < section.start);
sections.insert(index, section);

```

Then, for each piece and each time round, every clip that overlaps the piece adds a `Play` for the part of it inside the piece. `max` and `min` clamp the clip to the piece:

```

let begin = clip.start.max(piece_from);
let end = clip_end.min(piece_to);
if begin < end {
    arrangement.plays.push(Play {
        pattern: clip.pattern,
        start: start + (begin - piece_from),
        skip: begin - clip.start,
        length: end - begin,
    });
}

```

The audio thread knows nothing of sections

`Command::SetSong` carries the arrangement's `plays`, and all the audio thread does with them is a remainder:

```
pub fn pattern_step(&self, step: usize, pattern_length: usize) ->
Option<usize> {
    if pattern_length == 0 || !(self.start..self.end()).contains(&step) {
        return None;
    }
    Some((self.skip + step - self.start) % pattern_length)
}
```

`%` wraps the step around the pattern, which is all that “repeat” means. Everything about names, rounds and loops stays in `lineup-core`, where it can be tested without a sound card. The audio thread stays small, and it allocates nothing: the `Vec<Play>` is built on the UI thread, and the old one is sent back to be freed there.

The WAV export and the MIDI export use the same `Arrangement`, so all three agree on what the song is. The MIDI writer needs the opposite question: on which song steps does a note at pattern step 2 play? That is `Play::song_steps_of`, which uses an iterator with `step_by`:

```
(first..self.length)
    .step_by(pattern_length.max(1))
    .map(move |offset| self.start + offset)
```

Back from the song to the playlist

The audio thread counts song steps, so the playhead it reports is a song step. To draw it, the app turns it back into a timeline step with the arrangement's passes:

```
fn song_position(&self) -> Option<(usize, Pass)> {
    if self.mode != PlayMode::Song {
        return None;
    }
    let step = self.playhead()?;
    let song = self.project.playback_arrangement();
    let pass = *song.pass_at(step)?;
    Some((pass.from + (step - pass.start), pass))
}
```

The `Pass` travels with it to the playlist, which is how a section knows to show “► 2 of 2”. The channel rack, the piano roll and the harmony keys get the timeline step, so

they show the right step of the right clip during a repeat too.

Saving: format 4

`Clip` and `Section` both have a `repeats` field, and most of the time it is 1. `serde` can leave a field out of the file when it has its usual value, and fill it back in when reading a file that does not have it:

```
#[serde(default = "once", skip_serializing_if = "is_once")]
pub repeats: usize,
```

`default` names a function that returns the value to use, and `skip_serializing_if` a function that says when to leave the field out. The lists in the file (`sections`, `loop_region`) use the plain `#[serde(default)]` with `Vec::is_empty` and `Option::is_none`. So a project that uses none of this is written byte for byte as before, and an older Ferrous Lineup opens it.

An older version would also open a project that *does* use repeats: it skips fields it does not know. But it would play every repeat once, which is a song that sounds wrong without a word of warning. So `format_for` returns **4** for any project with sections, a loop region or a repeated clip, and older versions refuse it with a clear message ([Saving](#)).

Drawing it: a module inside a module

The strips are drawn in a file of their own. A file module can have children: `mod sections;` inside `playlist.rs` loads `playlist/sections.rs`. A child module can see its parent's private items, so `sections.rs` uses `Layout`, `Snap` and `PlaylistAction` from `playlist.rs` with `use super::{Layout, PlaylistAction, Snap};` without making them public to the rest of the app.

Each section is its own clickable area, made with `ui.interact` over a rectangle we choose, which returns a `Response` like any widget:

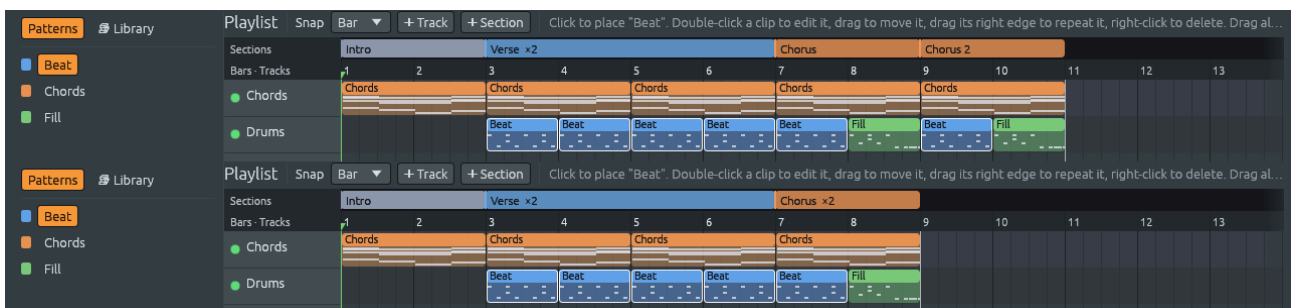
```
let response = ui.interact(rect, ui.scope_id().with(("section", index)),
                          Sense::click_and_drag());
```

The id must be different for each section and the same from one frame to the next, so it is built from the section's number. The menu is an `egui::Popup` attached to that response, opened on a click and closed only by a click outside, because it holds a text field you type into.

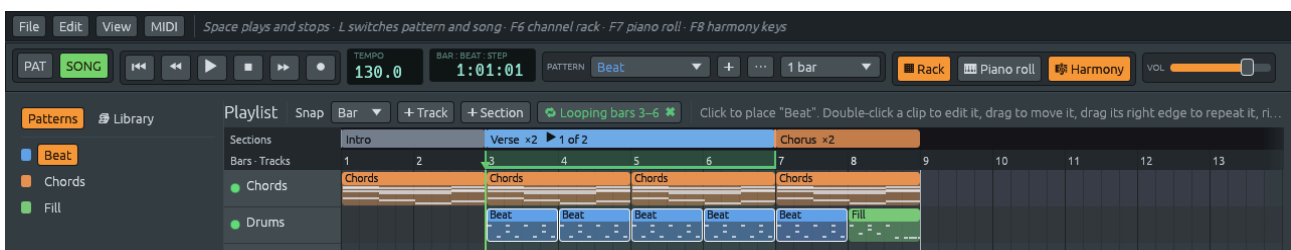
Dragging a section does not change the project on every frame, the way dragging a clip does. Only an outline is drawn, and the project changes once, when the button comes up. A section moves inside a sorted list: if one frame's move jumped it past a neighbour, its number would change in the middle of the drag, and the next frame would move the wrong section.

Writing out the repeats

Write out the repeats in a section's menu turns "Chorus ×3" into Chorus, Chorus 2 and Chorus 3, one after the other, each played once, so you can change the third chorus without changing the others. The song sounds the same before and after; a test checks it by comparing the two arrangements.



It shares its code with **Duplicate**: `make_room` moves everything after the section later, then the clips that start inside the section are copied into the gap. One thing to watch: clips are copied whole. The fill in the picture starts inside the chorus and runs past its end, so the original and its copy overlap. Ferrous Lineup cannot cut a clip in two (a clip has no "start this far into the pattern"); one of the [exercises](#) of this part adds it.



Telling the audio thread what changed

Files: `crates/lineup-core/src/diff.rs`, `crates/lineup-app/src/app/edit.rs`,
`crates/lineup-audio/src/engine.rs` and `crates/lineup-audio/src/sample.rs`

One way in for every change

The audio thread keeps its own copy of everything it plays: each channel's instrument, volume, pan and notes, the pattern lengths, the song (see [Two threads](#)). The window must keep that copy up to date, and a change can touch anything. A click on a step changes one list of notes. Deleting a pattern renumbers every clip after it. Typing a chord makes the band write new patterns. Undo replaces the whole project with an older one, which can differ in anything at all.

There are three ways to tell the audio thread about a change:

1. **Send the whole project again:** every pattern and channel, as when a file is opened. Simple, but every instrument would be built again, cutting off every note that is ringing, and a sample file would be read again for a one-step undo.
2. **Write, for each kind of edit, the commands it needs**, and for undo, the commands that reverse it. That is a hand-written pair for every edit, and every pair is a chance to get the reverse wrong (the [next chapter](#) says more about this choice).
3. **Compare the project before and after**, and send only what differs.

Ferrous Lineup does the third, for *every* change, not only for undo. No edit sends its own commands. Every change goes through one method, `edit`:

```

pub(super) fn edit<R>(
    &mut self,
    label: &'static str,
    change: impl FnOnce(&mut Project) -> R,
) -> R {
    if self.history.is_open() {
        // A drag or a typing session is under way, and its first change
        // made the undo step. Keep changing the project without copying
        // the whole of it every frame; the audio catches up steadily
        // from the last state it was given.
        let (patterns, channels) =
            (self.project.patterns().len(),
self.project.channels().len());
        let result = change(&mut self.project);
        self.project.follow_chord_sheet();
        self.file.project_edited();
        self.recovery.project_changed();
        self.step_unsynced = true;
        if self.project.patterns().len() != patterns
            || self.project.channels().len() != channels
        {
            // A pattern or channel came or went: the views and the
            // audio thread's rows have to move now, not in 50 ms.
            self.sync_step();
        } else {
            self.sync_step_if_due();
        }
        return result;
    }
    // A step left open by a direct history call: send it before the
    // audio's state is assumed to be this project.
    self.finish_step_sync();
    let before = self.project.clone();
    let result = change(&mut self.project);
    // The band plays what the chord sheet says now, in the same step.
    self.project.follow_chord_sheet();
    if self.project != before {
        self.sync_audio(&before);
        self.synced = self.project.clone();
        self.step_unsynced = false;
        self.last_sync = Instant::now();
        self.file.project_reverted();
        self.recovery.project_changed();
        // `before` is moved into the history here, after its last use.
        self.history.record(label, before);
    }
    result
}

```

Read the second half first, the ordinary case. `edit` keeps a copy of the project, lets the closure change it, and if anything changed, it does four things in one place:

- `sync_audio(&before)` tells the audio thread what differs. The rest of this chapter is about it.
- `follow_chord_sheet` lets the band write its patterns again if the chord sheet, the key or the bars changed, as part of the same step (see [How the band plays a chord sheet](#)).
- `file` and `recovery` hear that the project changed: the title bar gets its unsaved-changes star, and a recovery copy will be written (see [Saving](#)). `project_reverted` works the star out again from scratch, because a change made in one go can put back exactly what was saved, as a step clicked on and off again does.
- `history.record` keeps `before` as the undo step called `label` (see [Undo and redo](#)).

Having one path pays off twice. There is only one piece of code that turns project changes into commands, so it can be tested thoroughly on its own. And it runs on every click, so if it had a mistake you would hear it while editing, not only in the rare moment you undo something unusual.

A drag is one step, not one per frame

The first half, the `is_open` branch, is the important performance detail. A drag changes notes on every frame, sixty times a second, but only its first frame needs the copy for undo and the full comparison. While the step is open, the project is changed directly, and `synced` holds the last state the audio thread was given.

`sync_step_if_due` compares the project with that state and sends the differences at most every 50 ms:

```
/// How often, at most, a drag's changes reach the audio thread. The
/// window still draws every frame; only the sound waits this long, and
/// the whole project is copied only when this comes round.
const SYNC_INTERVAL: Duration = Duration::from_millis(50);

fn sync_step_if_due(&mut self) {
    if self.last_sync.elapsed() >= SYNC_INTERVAL {
        self.sync_step();
    }
}

fn sync_step(&mut self) {
    let before = std::mem::replace(&mut self.synced, Project::empty());
    self.sync_audio(&before);
    self.synced = self.project.clone();
    self.step_unsynced = false;
    self.last_sync = Instant::now();
}
```

`std::mem::replace` takes the old `synced` out of the struct and leaves an empty project in its place, so `sync_audio` can borrow it as `before` without a copy (it could not borrow `self.synced` while also borrowing `self` mutably). When a pattern or a channel comes or goes in the middle of a step, waiting is not good enough: the views count rows that the audio thread does not have yet, so `sync_step` runs at once.

When the mouse is released, `finish_step_sync` sends the last missing changes, and the step closes. One drag is one undo step, one copy and a handful of messages.

Undo and redo replace the whole project, so they always do the full comparison:

```
pub(super) fn undo(&mut self) {
    self.finish_step_sync();
    let before = self.project.clone();
    if self.history.undo(&mut self.project).is_some() {
        self.sync_audio(&before);
        self.synced = self.project.clone();
        self.step_unsynced = false;
        self.last_sync = Instant::now();
        self.file.project_reverted();
        self.recovery.project_changed();
        self.restore_song_library();
        self.keep_place_in_song(&before);
    }
}
```

The last two lines are for the window, not the sound: the library shows the style and rhythm this version of the project plays, and if undo changed the time signature, playback stays on the same bar.

Reading edit

`edit` takes a **closure** that changes the project, and the closure gets the project as its argument:

```
self.edit("move clip", |project| project.move_clip(index, track, start));
```

Why not let the closure reach `self.project` itself? Because `edit` already holds `&mut self`, and a closure that borrowed `self` too would be a second borrow of the same value while the first is still in use. The borrow checker refuses that. Handing the closure `&mut self.project` as an argument lends it exactly the part it needs, for exactly as long as the call lasts.

The closure's type is `impl FnOnce(&mut Project) -> R`. **FnOnce** is the closure trait for closures called exactly once: it lets the closure move a value into the project, such

as a new `channel`, because nothing will call it a second time. (`Fn` closures may be called many times and only read what they capture; `FnMut` ones may change it.)

`R` is whatever the closure returns, and `edit` passes it back to the caller. Many `Project` methods return something: `add_clip` returns the new clip's index as an `Option<usize>`, `remove_pattern` the pattern it removed, `insertBars` whether there was room. Because `R` can be any type, a closure can simply be the method call, `|project| project.add_clip(clip)`, and the caller gets the answer. If `edit` asked for `impl FnOnce(&mut Project)`, which must return `()`, each of those would need braces and a semicolon to throw the value away.

`edit_pattern` is a small wrapper for the most common case, a change to the selected pattern:

```
fn edit_pattern(&mut self, label: &'static str, change: impl FnOnce(&mut
Pattern)) {
    let index = self.pattern;
    self.edit(label, |project| {
        if let Some(pattern) = project.pattern_mut(index) {
            change(pattern);
        }
    });
}
```

`self.project != before` compares the two projects completely: the derived `PartialEq` compares every field, every list, every note. If nothing changed, such as a clip dragged back onto the same cell or a note deleted where there was none, nothing is sent and no undo step is recorded.

diff: the list of changes

The comparison lives in `lineup-core`, which knows projects but nothing about the audio thread. It describes differences in the project's own words:

```

pub enum Change {
    RemovePattern(usize),
    InsertPattern(usize),
    RemoveChannel(usize),
    InsertChannel(usize),
    Instrument(usize),
    Volume(usize),
    Muted(usize),
    Pan(usize),
    ReverbSend(usize),
    Effects(usize),
    PatternLength(usize),
    Notes { pattern: usize, channel: usize },
    Tempo,
    LooseTiming,
    Mixer,
    Song,
}

pub fn diff(before: &Project, after: &Project) -> Vec<Change>

```

The changes are steps to follow **in order**, and each number is a position at the moment its change is applied. After `RemovePattern(0)`, the pattern that used to be number 1 is number 0, and any later change calls it 0. Values (a volume, a list of notes) are not in the `Change`: they are read from the newer project when the change is turned into a command. That keeps `Change` small and `Copy`.

Only what makes a sound is compared. Pattern names, channel names, track names, section names and the key give no change at all, because the audio thread never sees them. Muting a track does give one: it takes that track's clips out of the song, so the clips the audio thread plays change.

The mixer is audible too. Moving a channel between the speakers gives `Pan`, changing how much of it goes to the reverb gives `ReverbSend`, changing its EQ, compressor or delay gives `Effects`, and changing the master volume or the reverb itself gives `Mixer`. Soloing a channel is a mixer change *and* changes which channels are heard, so `diff` also reports `Muted` for every channel whose effective mute changed. Loose timing gives `LooseTiming`: it decides how far off the grid a note may start.

Lining up two lists

The hard part is the lists. Say the patterns were `[Beat, Chords, Fill]` and now are `[Beat, Fill]`. Comparing item by item would say pattern 1 changed from Chords to Fill and pattern 2 disappeared, which is true but clumsy: it would send all of Fill's notes again. What really happened is "pattern 1 was removed".

`Alignment` finds that. It counts how many items are the same at the start of both lists, and how many at the end. What is left in the middle was removed from the old list and inserted in the new one:

before:	Beat		Chords		Fill		start = 1 (Beat is the same)
after:	Beat				Fill		end = 1 (Fill is the same)
							removed = 3 - 1 - 1 = 1
							inserted = 2 - 1 - 1 = 0

```
fn new<T>(before: &[T], after: &[T], same: impl Fn(&T, &T) -> bool) ->
Self {
    if before.len() == after.len() {
        return Alignment { start: 0, removed: 0, inserted: 0 };
    }
    // The same at the start...
    let start = before
        .iter()
        .zip(after)
        .take_while(|(old, new)| same(old, new))
        .count();
    // ...and the same at the end, without counting an item twice.
    let room = before.len().min(after.len()) - start;
    let end = before[start..]
        .iter()
        .rev()
        .zip(after[start..].iter().rev())
        .take_while(|(old, new)| same(old, new))
        .count()
        .min(room);
    Alignment {
        start,
        removed: before.len() - start - end,
        inserted: after.len() - start - end,
    }
}
```

The iterator chain reads well once you know the pieces: `zip` walks two lists side by side, giving pairs; `take_while` keeps pairs while they are equal and stops at the first that is not; `count` says how many were kept. `rev` walks the lists backwards, for the end.

`.min(room)` is easy to miss and important. With `[A, A]` becoming `[A]`, the start finds one equal item, and so does the end, *the same* `A`. Both counts together would say two items are kept out of a list of one, and `after.len() - start - end` would be `1 - 1 - 1`, which for a `usize` is not -1 but a panic (“attempt to subtract with overflow”). Limiting the end to what is left after the start keeps the arithmetic honest.

When both lists are as long, nothing was added or removed, so items are simply compared position by position. That covers the most common edits: a click on a step changes one pattern, in place.

`Alignment` works for any `T`, because the caller says what “the same” means with the `same` closure. For channels that is full equality: `|old, new| old == new`. For patterns it is **not**, and that is the subtle part. A pattern’s rows change when channels come and go: when the band joins, every pattern gets a row for each of its channels, so comparing whole patterns would say every pattern changed. The alignment would then send every pattern’s notes again, and the app would select a pattern that only looks new.

So patterns are lined up by the things that identify them, not by their contents:

```
/// Whether two patterns are the same pattern, perhaps with other notes.
fn same_pattern(old: &Pattern, new: &Pattern) -> bool {
    old.name == new.name && old.color == new.color && old.player ==
new.player
}
```

The notes and the length are what the aligner lets change in place. Name, colour and player say *which* pattern it is; the notes that differ are sent afterwards as `Notes` and `PatternLength` changes anyway.

fn pointers: enum variants are functions

Once two lists are lined up, the changes are: remove the middle of the old list, then insert the middle of the new one.

```
fn changes(
    self,
    remove: fn(usize) -> Change,
    insert: fn(usize) -> Change,
) -> impl Iterator<Item = Change> {
    let removals = (self.start..self.start +
self.removed).rev().map(remove);
    let insertions = (self.start..self.start + self.inserted).map(insert);
    removals.chain(insertions)
}
```

and it is called like this:

```
patterns.changes(Change::RemovePattern, Change::InsertPattern)
```

`Change::RemovePattern` is not a value here, it is a **function**. Every enum variant that holds data in parentheses is also a function that builds it: `Change::RemovePattern` has the type `fn(usize) -> Change`. The lowercase `fn(usize) -> Change` is a **function pointer** type: the address of a plain function, with no captured variables. A closure that captures nothing can be one too. `map(remove)` calls it for every number, building `RemovePattern(2)`, `RemovePattern(1)` and so on.

Removals go from the **last** one backwards. Removing pattern 2 first leaves patterns 0 and 1 where they were, so `RemovePattern(1)` still means the right pattern. Going forwards, removing pattern 1 first would move pattern 2 down to 1, and the second removal would hit the wrong one. Insertions go forwards for the same reason: once position 1 is filled, position 2 is right after it.

`chain` joins two iterators into one: first all of `removals`, then all of `insertions`. The function returns `impl Iterator<Item = Change>`, "some iterator of changes", without naming its long real type (`Chain<Map<Rev<Range<usize>>, fn(usize) -> Change>, ...>`).

The order of everything

```
pub fn diff(before: &Project, after: &Project) -> Vec<Change> {
    let patterns = Alignment::new(before.patterns(), after.patterns(),
    same_pattern);
    let channels = Alignment::new(before.channels(), after.channels(),
    |old, new| old == new);

    let mut changes: Vec<Change> = patterns
        .changes(Change::RemovePattern, Change::InsertPattern)
        .chain(channels.changes(Change::RemoveChannel,
    Change::InsertChannel))
        .collect();
    ...
}
```

Patterns come first, then channels, then everything inside them. The order matters because of what the audio thread does with each change:

- An inserted **pattern** arrives in the audio thread with no notes in any channel; its notes follow as `Notes` changes.
- An inserted **channel** arrives complete: its instrument, volume, and its notes in every pattern. So the patterns must already be right when it comes, or it would bring notes for the wrong number of patterns.

Then, for every channel that existed before (`old_index` says where it was, or `None` if it is new), the instrument and mix are compared:

```

for (index, channel) in after.channels().iter().enumerate() {
  let Some(old_index) = channels.old_index(index) else {
    continue; // a new channel arrives complete
  };
  let old = &before.channels()[old_index];
  if old.kind != channel.kind
    || old.program != channel.program
    || old.sent_on != channel.sent_on
  {
    changes.push(Change::Instrument(index));
  }
  if old.volume != channel.volume {
    changes.push(Change::Volume(index));
  }
  if old.pan != channel.pan {
    changes.push(Change::Pan(index));
  }
  if old.reverb_send != channel.reverb_send {
    changes.push(Change::ReverbSend(index));
  }
  if old.effects != channel.effects {
    changes.push(Change::Effects(index));
  }
  if before.channel_muted(old_index) != after.channel_muted(index) {
    changes.push(Change::Muted(index));
  }
}

```

The instrument is built from three fields: the kind (drum, sample file, Lineup Osc, MIDI Out), the General MIDI program, and `sent_on`, the MIDI port a drum is sent to instead of playing here. `channel_muted` is not simply the channel's mute button: it also takes the mixer's solo into account. That is why a solo change shows up as a `Muted` change on the other channels, as well as a `Mixer` change.

`let ... else` either takes the value out of the `Some`, or runs the `else` block, which must leave: here `continue` jumps to the next channel.

Then the notes, for every pattern and every old channel:


```

for (index, pattern) in after.patterns().iter().enumerate() {
  let old_index = patterns.old_index(index);
  let old_pattern = old_index.map(|old| &before.patterns()[old]);
  if let Some(old) = old_pattern
    && old.length() != pattern.length()
  {
    changes.push(Change::PatternLength(index));
  }
  for channel in 0..after.channels().len() {
    let Some(old_channel) = channels.old_index(channel) else {
      continue; // a new channel brought its notes with it
    };
    // The notes as they are played, with the song's swing: a
    // new amount, or a pattern now in a time signature that does
    // not swing, plays other notes. A new pattern starts with no
    // notes in the audio thread.
    let old_notes = match old_index {
      Some(old) => before.played_notes(old, old_channel),
      None => Cow::Borrowed(&[][..]),
    };
    if old_notes != after.played_notes(index, channel) {
      changes.push(Change::Notes { pattern: index, channel });
    }
  }
}

```

The `if let ... && ...` is a **let chain**: the block runs only if the pattern is old *and* its length changed.

The notes compared are `played_notes`, not the written ones. Swing moves the offbeat notes later without changing what is written (see [Tempo, meter, swing and feel](#)), and the audio thread must be sent what is *heard*. So turning the swing up changes no written note, but gives a `Notes` change for every list it moves.

`played_notes` returns a `Cow` (“clone on write”): a `Cow::Borrowed` when the written notes are played as they are, which costs nothing, and a `Cow::Owned` with a changed copy only when swing moves them. Two `Cow`s compare by their contents, whichever kind they are.

`&[][..]` is an empty slice: `[]` is an empty array, and `[..]` takes all of it as a slice. A new pattern is compared with “no notes”, which is exactly what the audio thread has for it, so a `Notes` change is sent only for the channels that have notes in it.

Last come the tempo, the looseness, the mixer and the song. The song is compared through the arrangement that song mode actually plays:

```

let (old_song, new_song) = (before.playback_arrangement(),
after.playback_arrangement());
if old_song.plays != new_song.plays
  || old_song.length != new_song.length
  || before.played_tempos(&old_song) != after.played_tempos(&new_song)
{
  changes.push(Change::Song);
}

```

An `Arrangement` is the clips with their repeats and the sections' repeats already unrolled, cut to the loop region if there is one (see [The project](#)). Comparing it catches a moved clip, a mute, a repeat count, a changed loop and a tempo marker in one `Song` change. Tempo markers are part of the song because the length of a step changes partway through it.

When the story is ambiguous

Say there are three Lineup Osc channels with the same settings but different notes, and you delete the middle one. Comparing the *channels* alone (name, instrument, volume, mute), the old list is `[0sc, 0sc, 0sc]` and the new one `[0sc, 0sc]`, and the alignment cannot tell which one went: it finds two equal at the start and says the *last* was removed.

That is fine. The notes of the second channel now differ (it holds what used to be the third channel's notes), so a `Notes` change follows, and the audio thread ends up with exactly the right channels and notes. The diff does not need to guess what you really did. It needs a list of steps that ends in the same place, and a short one where it can.

Testing with a model

How do you test a function like `diff`? Checking its exact output for every case would be brittle, and would not prove the output is *right*. What we really want to know is: *if the audio thread follows these changes, does it end up with the new project?*

So the tests build a small **model** of what the audio thread knows: the pattern lengths, each channel's instrument, mix, effects and notes per pattern, the tempo, and the song. It is simple enough to trust at a glance:

```

struct ModelChannel {
    kind: ChannelKind,
    program: Option<u8>,
    volume: f32,
    muted: bool,
    pan: f32,
    reverb_send: f32,
    effects: crate::Effects,
    /// One list per pattern.
    notes: Vec<Vec<NoteEvent>>,
}

struct Model {
    lengths: Vec<usize>,
    channels: Vec<ModelChannel>,
    tempo: Tempo,
    loose_timing: u8,
    plays: Vec<Play>,
    song_length: usize,
    tempos: Vec<Tempo>,
}

```

`Model::apply` does to the model what the audio thread does for each change, and `check` puts it together:

```

/// Diffs two projects, checks that following the changes turns the
/// first into the second, and returns the changes.
fn check(before: &Project, after: &Project) -> Vec<Change> {
    let changes = diff(before, after);
    let mut model = Model::of(before);
    for &change in &changes {
        model.apply(change, after);
    }
    assert_eq!(model, Model::of(after), "changes: {changes:?}");
    changes
}

/// Checks both directions, as undo and then redo would.
fn check_both_ways(before: &Project, after: &Project) -> Vec<Change> {
    check(after, before);
    check(before, after)
}

```

Every test checks this property first, and only then, where it matters, the exact changes: that a step click gives a single `Notes`, that names and the key give nothing, that undoing a deleted pattern starts with `InsertPattern(0)`, that swing changes the notes played. The last test makes a mess on purpose, removing a pattern, adding a channel, toggling a step, changing a volume and the tempo all at once, and also diffs an empty project against the demo song, as opening a file would.

This kind of test is called **property-based**: it checks a rule that must hold for any input, rather than one answer for one input. One of the [exercises](#) takes it further, with projects changed at random.

The app's side: `sync_audio`

`sync_audio` turns each `Change` into a command, reading values from the current project:

```

fn sync_audio(&mut self, before: &Project) {
    let sample_rate = self.sample_rate();
    let mut patterns_moved = false;
    let mut channels_changed = false;
    for change in diff(before, &self.project) {
        match change {
            Change::RemovePattern(index) => {
                self.send(Command::RemovePattern(index));
                // Keep the same pattern selected. If it was the one
                // removed, the next one takes its place.
                if index < self.pattern {
                    self.pattern -= 1;
                }
                patterns_moved = true;
            }
            Change::InsertPattern(index) => {
                let pattern = &self.project.patterns()[index];
                let (length, bands) = (pattern.length(),
pattern.player.is_some());
                self.send(Command::InsertPattern { index, length });
                if bands {
                    // A part the band wrote, as you typed a chord:
                    // the pattern you had stays selected.
                    if index <= self.pattern {
                        self.pattern += 1;
                    }
                } else {
                    // A new pattern (added, cloned, or brought back
                    // by undo) is the one you want to see, from its
                    // start.
                    self.pattern = index;
                    self.rewind_for_new_pattern();
                }
                patterns_moved = true;
            }
            Change::Notes { pattern, channel } => {
                let notes = self.project.played_notes(pattern,
channel).into_owned();
                self.send(Command::SetNotes { pattern, channel, notes });
            }
            ...
        }
    }
    ...
}

```

The other arms are just as direct. `InsertChannel` builds the channel's instrument and sends it with its notes for every pattern. `Instrument` builds a new instrument and sends `SetInstrument`. `Volume`, `Pan`, `ReverbSend` and `Effects` send their one value, `Muted` sends the channel's effective mute, `LooseTiming` sends the new amount, `Mixer` sends the master volume and the reverb settings, and `Song` sends the three

commands that replace the song, its tempos and its metronome clicks. `Tempo` sends the tempo at the practice speed, if you slowed the song down to practise (see [Practice tools](#)).

`into_owned` turns the `Cow` from `played_notes` into a `Vec` the command can carry: it clones a borrowed list, and simply hands over an owned one.

`sync_audio` also keeps the window's own numbers pointing at the right things. The selected pattern is a position, so it moves down when a pattern before it goes, and up when the band inserts one before it. The piano roll remembers which channel it shows, so it is told when channels come and go (`channel_removed` and `channel_inserted`). Because these updates are here, adding, cloning and deleting patterns select the right pattern whether you did them or undo did.

The audio thread inserts in place

Undoing “delete pattern 0” must put a pattern back at position 0, not at the end. So the audio thread's commands take a position:

```
/// Add an empty pattern at position `index`; the patterns from there
/// on move up one place. An index past the end adds it at the end.
InsertPattern { index: usize, length: usize },

/// Add a channel at position `index`, with its notes in every pattern.
/// The channels from there on move down one row. An index past the
/// end adds it at the bottom.
InsertChannel {
    index: usize,
    instrument: Instrument,
    notes: ChannelNotes,
    volume: f32,
    muted: bool,
},
```

`Vec::insert(index, value)` moves every item from `index` on one place up, and puts `value` in the gap. Would that allocate on the audio thread? Only if the `Vec` were full. Room for `MAX_CHANNELS` channels and `MAX_PATTERNS` patterns is reserved when the audio thread is made (see [Room reserved in advance](#)), and both commands check the limit first, so `insert` always fits in memory that is already there. Moving items along is just copying, which is allowed in real time:

```

fn insert_pattern(&mut self, index: usize, length: usize) {
    if self.pattern_lengths.len() < MAX_PATTERNS {
        let index = index.min(self.pattern_lengths.len());
        self.pattern_lengths.insert(index, length);
        for channel in &mut self.channels {
            channel.notes.insert_empty(index);
        }
        ...
    }
}

```

An index past the end is clamped (`index.min(len)`) rather than refused. The audio thread must never panic, and “insert after the last one” is the most sensible reading of a too-big number.

`InsertChannel` is one of the biggest commands. Every slot of the command queue is as big as the biggest command, so a test keeps commands at 80 bytes at most. An instrument’s voices are behind a `Box`, so a command carries a pointer to them, never all of them.

Instruments come back from a cache

Undoing “load sample” gives a channel its old instrument back, and that might be a sample file. Reading a file on the window’s thread for an undo would make the window stutter, and the file might even be gone by now.

So the app keeps a `SampleCache`: every sample file read so far, by path.

```

pub struct SampleCache {
    loaded: HashMap<PathBuf, Arc<SampleData>>,
}

impl SampleCache {
    pub fn get(&mut self, path: &Path) -> Result<Arc<SampleData>,
SampleError> {
        // A `HashMap<PathBuf, _>` can be searched with a `&Path`, because
        // `PathBuf` implements `Borrow<Path>`: no `PathBuf` is built
here.
        if let Some(sample) = self.loaded.get(path) {
            return Ok(Arc::clone(sample));
        }
        self.reload(path)
    }

    pub fn reload(&mut self, path: &Path) -> Result<Arc<SampleData>,
SampleError> {
        let sample = Arc::new(SampleData::load_wav(path)?);
        self.loaded.insert(path.to_path_buf(), Arc::clone(&sample));
        Ok(sample)
    }
}

```

A **HashMap** finds a value by its key in about the same time however many entries it holds. Its keys are owned **PathBuf**s, but **get** takes a borrowed **&Path**. That works because of the **Borrow trait**: **PathBuf** implements **Borrow<Path>**, a promise that a **PathBuf** and the **Path** it holds hash and compare the same way. So the map can look up a **&Path** directly, without building a **PathBuf** just to search. (**HashMap<String, _>** can be searched with a **&str** for the same reason.)

The cache hands out **Arc** clones, so the cache and every sampler using the sound share one copy of it in memory. A test checks this with **Arc::ptr_eq**, which asks whether two **Arc**s point at the very same memory, not merely at equal values:

```

assert!(Arc::ptr_eq(&first, &second), "the second get shares the first");
assert!(!Arc::ptr_eq(&first, &reloaded), "reload reads the file again");

```

Dropping a file on the rack calls **reload**, not **get**: you may have changed the file in an audio editor since, and dropping it again should pick that up. A file that cannot be read is not remembered, so the next attempt reads it again. Opening another project clears the cache.

Instrument::for_channel takes the cache as an argument (**&mut SampleCache**) instead of reading files itself. Exporting builds every instrument again (see [Exporting](#)), and thanks to the cache it reads no files either.

Undo and redo

Files: `crates/lineup-core/src/history.rs` and `crates/lineup-app/src/app/edit.rs`

Every change to a song can be undone with Ctrl+Z and redone with Ctrl+Shift+Z or Ctrl+Y, and the Edit menu names the step it would undo (“Undo move clip”). This chapter is about how the history is kept, and when one step ends and the next begins.

Two ways to undo

There are two classic ways to build undo.

The **command pattern** records every change as an instruction that knows how to reverse itself: “add note C4 at step 3” is undone by “remove note C4 at step 3”. The history is a list of instructions. It uses very little memory, but every kind of change needs its own reverse, written by hand, and each one is a new place to make a mistake. Think of what undoing “delete pattern” would need: put the pattern back at its old position, with its name and length and the notes of every channel, put back every clip that used it, and renumber the clips of the patterns after it again. If one detail is forgotten, undo quietly gives back a slightly different song.

Snapshots keep a whole copy of the project from before each undoable step. Undo puts the copy back. There is nothing to reverse, so there is nothing to reverse wrongly: whatever the change did, the copy is exactly what was there before. The cost is memory, and the time to make a copy.

Ferrous Lineup uses snapshots. A `NoteEvent` takes 24 bytes, and a song has a few hundred or a few thousand of them, so a copy of a whole project is some kilobytes. Even a big project with 10,000 notes holds about 240 KB of notes, and making one copy takes far less than a millisecond. The app keeps at most 256 MiB of snapshots: a small song gets the full 100 steps, while a very large one still gets at least 10 (which can go over the budget). That is a fair price for undo that is correct by construction. (One of the [exercises](#) shows how to make the copies much cheaper, if a project ever gets big enough to need it.)

History<T>

The history lives in `lineup-core`, and it does not know about projects at all:

```
struct Step<T> {
    label: &'static str,
    state: T,
}

pub struct History<T> {
    /// The oldest step is at the front, the newest at the back. A
    /// `VecDeque` can drop the oldest step from the front cheaply.
    undo: VecDeque<Step<T>>,
    redo: Vec<Step<T>>,
    limit: usize,
    /// Whether the newest undo step still takes in new changes.
    open: bool,
}

impl<T> History<T> {
    pub fn new(limit: usize) -> Self { ... }
    pub fn record(&mut self, label: &'static str, before: T) { ... }
    pub fn is_open(&self) -> bool { ... }
    pub fn close(&mut self) { ... }
    pub fn reopen(&mut self, label: &'static str) -> bool { ... }
    pub fn undo(&mut self, current: &mut T) -> Option<&'static str> { ... }
}

pub fn redo(&mut self, current: &mut T) -> Option<&'static str> { ... }

pub fn undo_label(&self) -> Option<&'static str> { ... }
pub fn redo_label(&self) -> Option<&'static str> { ... }
pub fn clear(&mut self) { ... }
}
```

`History<T>` is **generic**: `T` is whatever is being remembered. The app uses a `History<Project>`; the tests use a `History<i32>`, which makes them short and easy to read:

```

#[test]
fn changes_join_the_open_step() {
    let mut history = History::default();
    let mut value = 0;
    // A drag: three changes before the mouse button is released.
    for new in [1, 2, 3] {
        history.record("drag", value);
        value = new;
    }
    history.close();
    history.undo(&mut value);
    assert_eq!(value, 0, "one undo goes back to before the drag");
}

```

Look at the `impl` line: `impl<T> History<T>`, with no **trait bound** on `T`. Not even `Clone`. The history never copies a `T`; it only moves the values it is given from one list to another. The caller makes the copy (`self.project.clone()`) and hands it over. If a method of `History` ever called `.clone()` on a state, the compiler would refuse with “no method named `clone` found for type parameter `T`”, and we would have to write `impl<T: Clone>`. A generic type with no bounds is a strong promise: the code cannot do anything with a `T` except store it, move it and drop it.

The compiler makes a separate copy of the code for each `T` it is used with (this is **monomorphization**), so a `History<Project>` is as fast as if it had been written for `Project` by hand.

&'static str labels

Each step has a label, such as `"move clip"`, for the Edit menu. Its type is `&'static str`: a reference to text that lives for the whole program. Every string literal written in the code has this type, because the text is stored inside the program file itself. So a label costs nothing: no `String` is built, nothing is allocated, and the history holds no borrow that could end too soon. The price is that a label cannot be made at run time (“rename pattern Beat”), since that text would not exist until the program builds it.

The labels are written in English, and the Edit menu translates them when it shows them, so the history does not need to know the app’s language (see [Two languages](#)).

VecDeque: a list with two cheap ends

When there are already `limit` steps, recording one more forgets the oldest. In a `Vec`, removing the first item (`remove(0)`) moves every other item one place down. A **VecDeque** (“double-ended queue”, said “vec deck”) is a ring buffer, like the queues

between the threads (see [Two threads](#)): it keeps track of where its front is, so `push_back` and `pop_front` are both cheap.

The limit is not a constant in the app. `History::new` takes it, and `undo_limit_for` works it out from the project's size:

```
const UNDO_BYTES: usize = 256 * 1024 * 1024;
const MIN_UNDO_STEPS: usize = 10;

pub(super) fn undo_limit_for(project: &Project) -> usize {
    limit_for_bytes(project.heap_bytes())
}

fn limit_for_bytes(bytes: usize) -> usize {
    (UNDO_BYTES / bytes.max(1)).clamp(MIN_UNDO_STEPS, DEFAULT_LIMIT)
}
```

`heap_bytes` counts the notes, clips, markers and names, the things whose number grows with the song. A small project divides 256 MiB by a few kilobytes and gets more than `DEFAULT_LIMIT`, so it is clamped to 100. A huge project gets fewer, but never fewer than 10, so undo does not vanish just because the song is long. Opening a file builds a fresh `History` with the limit for that project. `History::default()` means 100, which is what the little tests use.

```
pub fn record(&mut self, label: &'static str, before: T) {
    self.redo.clear();
    if self.open {
        return;
    }
    self.undo.push_back(Step { label, state: before });
    if self.undo.len() > self.limit {
        self.undo.pop_front();
    }
    self.open = true;
}
```

Every new change clears the redo list. After you undo three steps and then change something, those three steps belong to a future that no longer exists. (A few programs keep a whole tree of histories, so no branch is ever lost; most simply drop the old future, as this one does.)

The redo list is a plain `Vec`, because steps are only added and taken at its end.

mem::swap: undo without copying

```
pub fn undo(&mut self, current: &mut T) -> Option<&'static str> {
    let mut step = self.undo.pop_back()?;
    // Now `current` holds the old state and `step` the newer one.
    std::mem::swap(current, &mut step.state);
    let label = step.label;
    self.redo.push(step);
    self.open = false;
    Some(label)
}
```

`pop_back()` gives an `Option`, and the `?` operator works on `Option` in a function that returns an `Option`: if there is nothing to undo, `undo` returns `None` at once and changes nothing.

`std::mem::swap` exchanges the values behind two `&mut` references. After it, `current` (the app's project) holds the older version, and the `step` holds the version we just left, ready for redo. Nothing is copied or allocated: a `Project` is mostly `Vec`s, and a `Vec` is a pointer, a length and a capacity, so swapping two projects swaps a few of those small values, whatever the size of the song. The borrow checker is what makes this simple: `swap` needs two `&mut` references, and two `&mut` references can never point at the same value, so `swap` can exchange the two values' bytes without checking anything.

`redo` is the mirror image, popping from `redo` and pushing onto `undo`.

When does one step end?

Dragging a clip across the playlist moves it a little on many frames, and each move is an edit. If each one were its own step, undoing the drag would take as many presses of Ctrl+Z as there were frames. So the history keeps the newest step **open**, and `record` does nothing while it is: the step already holds the project from before the drag started, which is exactly where undo should go back to.

The app decides when a step ends, because only it knows what the mouse is doing. It calls the history at the start and at the end of every frame:

```

/// Called at the start of each frame. A press of a mouse button
/// always begins a new undo step.
pub(super) fn start_frame_for_undo(&mut self, ctx: &egui::Context) {
    if ctx.input(|input| input.pointer.any_pressed()) {
        self.close_step();
    }
}

/// Called at the end of each frame. While a mouse button is held (a
/// drag), a text field has the keyboard (typing a name) or a take is
/// being recorded, changes join one undo step; once all of them are
/// over, the step ends.
pub(super) fn end_frame_for_undo(&mut self, ctx: &egui::Context) {
    let holding = ctx.input(|input| input.pointer.any_down());
    if !holding && !ctx.text_edit_focused() && !self.is_recording() {
        self.close_step();
    } else if self.step_unsynced {
        // The step goes on, but its last change may still be waiting
        // for its turn: a drag held still, or the last key typed, has
        // no next edit to carry it to the audio thread. Send it when
        // it is due, and make sure a frame comes then.
        self.sync_step_if_due();
        if self.step_unsynced {
            ctx.request_repaint_after(SYNC_INTERVAL.saturating_sub(self.last_sync.elapsed()));
        }
    }
}

```

Follow a few gestures through these two functions:

- **A click on a step.** egui reports a click on the frame the button is let go. That frame's edit opens a step, and the end of the same frame closes it, since no button is down any more: one change, one step.
- **A drag.** The press closes whatever was open. Every frame of the drag edits the project while the button is down, so the edits join one step. The frame where the button is let go closes it.
- **Typing a pattern name.** While the text field has the keyboard focus, every letter joins the same step. Clicking elsewhere takes the focus away and closes it, so undo puts back the whole old name at once.
- **Typing a tempo.** Clicking the BPM box turns it into a text field, so the same rule applies: everything typed there is one step, which ends when you leave the box.
- **Recording a take.** Every note played while recording joins one step, so one undo takes the whole take away.

`start_frame_for_undo` matters for one case: a click that starts while a text field still has the focus. The press closes the typing step, so the click's change does not join it.

The `else if` branch is about sound, not undo. While a step is open, changes reach the audio thread at most every 50 ms, carried by the next edit (see [Telling the audio thread what changed](#)). A drag held still makes no next edit, so the frame sends what is still waiting once it is due, and `request_repaint_after` asks egui for a frame at that moment even if nothing moves.

Closing a step is `close_step`:

```
fn close_step(&mut self) {
    self.finish_step_sync();
    if self.history.is_open() {
        self.file.project_reverted();
    }
    self.history.close();
}
```

It sends whatever the audio thread is still missing, and works out the unsaved-changes star again (a note dragged away and back leaves the song as it was saved). Then the history stops taking changes into the step.

Several clicks, one step

Some changes come one click at a time but belong together. Tapping a tempo is a click per beat; nudging the master volume or the number of choruses is a click or a drag per nudge. Undo should take all of them back at once. `reopen` lets the app open the newest step again:

```
/// Opens the newest step again, if it is called `label` and nothing
/// was undone since, so the next change joins it: taps of a tempo,
/// each a click of its own, make one step. Returns whether it did.
pub fn reopen(&mut self, label: &'static str) -> bool {
    if self.redo.is_empty() && self.undo_label() == Some(label) {
        self.open = true;
    }
    self.open
}
```

and the app calls it just before the edit:

```
pub(super) fn set_master_volume(&mut self, master: f32) {
    const STEP: &str = "master volume";
    self.history.reopen(STEP);
    self.edit(STEP, |project| project.mixer.master = master);
}
```

If the last step was another change, or something was undone since, the label does not match and a new step starts as usual.

Everything goes through edit

The pattern name field shows how strict the rule “every change goes through `edit`” is. `egui`’s text field wants a `&mut String` to change. Giving it the project’s own name would change the project directly, behind the history’s back: no undo step, and no message to the audio thread either. So the field edits a copy, and the copy goes through `edit` when it changed:

```
// The field edits a copy, so the change can go through `edit`.
let mut name = self.project.patterns()[self.pattern].name.clone();
if ui.text_edit_singleline(&mut name).changed() {
    self.edit_pattern("rename pattern", |pattern| pattern.name = name);
}
```

The `Project` type helps here: its lists are private, and only its methods can change them (see [The project](#)). Anything that changes a project needs a `&mut Project`, and in the app the only place that hands one out is `edit`.

Keyboard shortcuts

```
const UNDO: KeyboardShortcut = KeyboardShortcut::new(Modifiers::COMMAND,
Key::Z);
const REDO: KeyboardShortcut =
    KeyboardShortcut::new(Modifiers::COMMAND.plus(Modifiers::SHIFT),
Key::Z);
/// The other common shortcut for redo, used on Windows.
const REDO_Y: KeyboardShortcut = KeyboardShortcut::new(Modifiers::COMMAND,
Key::Y);

pub(super) fn edit_shortcuts(&mut self, ctx: &egui::Context) {
    let pressed = |shortcut| ctx.input_mut(|input|
input.consume_shortcut(&shortcut));
    // Ctrl+Z would also match Ctrl+Shift+Z, so redo is checked first.
    if pressed(REDO) || pressed(REDO_Y) {
        self.redo();
    } else if pressed(UNDO) {
        self.undo();
    }
}
```


`Modifiers::COMMAND` is Ctrl on Linux and Windows and `⌘` on a Mac. `egui's consume_shortcut` is forgiving about extra keys: a shortcut without Shift also matches when Shift is held. That is usually what you want (on many keyboards `+` needs Shift, and Ctrl++ should still work), but here it means Ctrl+Shift+Z would count as Ctrl+Z too. Checking redo first, and using `else if`, settles it.

`||` stops at the first `true`, so if Ctrl+Shift+Z was pressed, Ctrl+Y is not even looked at. “Consume” means the key press is taken out of `egui's` input, so nothing else in the same frame reacts to it.

These shortcuts are skipped while a text field has the keyboard. `egui's` text fields have their own undo for typing, and Ctrl+Z there should undo the last letters, not the last change to the song.

The Edit menu shows the same shortcuts, written the way the system writes them (“Ctrl+Z” on Linux, “⌘Z” on a Mac), and puts the translated label in the item, “Undo move clip”.

Undo and the unsaved-changes star

The window title shows a star when the project differs from the saved file. It does not compare the two projects on every frame: an edit sets a flag, and the comparison happens when the title is next worked out. Undo and redo set the flag too, so they can make the star go away when the project is equal to the saved one again. A plain `dirty: bool` set on every edit would have kept the star after the undo. [Saving](#) shows the rest of this.

Opening a file, or File > New, replaces the whole project and clears the history. Undoing into a different song would be surprising, and the “Save changes?” question already protects what you leave behind.

Saving: the .lineup file, recovery and damaged files

Files: `crates/lineup-core/src/file.rs`, `crates/lineup-app/src/app/files.rs`, `crates/lineup-app/src/app/recovery.rs`, `crates/lineup-app/src/app/recent.rs` and `crates/lineup-core/tests/damaged_files.rs`

A song is saved as a `.lineup` file. This chapter is about what is in that file and why it is shaped that way, how a file from an older or a newer version is read, how saving avoids ever leaving a half-written file, and how the app keeps a copy of unsaved work in case it crashes. The numbered list of format versions is in the reference page [Project file format](#).

What a project file looks like

A `.lineup` file is plain text. To see one, open a demo under File > Demos, use File > Save As..., and open the saved file in a text editor. File > New beat starts a song with an empty pattern and a drum kit; this is the whole file it saves:

```
(
  format: 1,
  tempo: 130.0,
  scale: (
    root: C,
    kind: Major,
  ),
  channels: [
    (
      name: "Kick",
      kind: Sampler(Builtin(Kick)),
      volume: 0.8,
      muted: false,
    ),
    (
      name: "Clap",
      kind: Sampler(Builtin(Cl原因)),
      volume: 0.8,
      muted: false,
    ),
    ...
    (
      name: "Lineup Osc",
      kind: Synth,
      volume: 0.6,
      muted: false,
    ),
  ],
  patterns: [
    (
      name: "Pattern 1",
      length: 16,
      notes: [
        [],
        [],
        [],
        [],
        [],
      ],
    ),
  ],
  tracks: [
    (
      name: "Track 1",
      muted: false,
    ),
  ],
  clips: [],
)
```

Other songs carry more. The starting song's synth channel plays a General MIDI piano, so it carries `program: Some(0)` and its file says `format: 9`. Its chords read as notes,

one per line:

```
(pitch: 48, start: 0, length: 8, velocity: 100),
(pitch: 60, start: 0, length: 8, velocity: 100),
(pitch: 64, start: 0, length: 8, velocity: 100),
(pitch: 67, start: 0, length: 8, velocity: 100),
```

File > New, an empty chord sheet with a band, saves `format: 17` and ends with the band: `band: Some((style: "Pop beat", ...))`.

The format is **RON**, “Rusty Object Notation”. It looks like Rust: a struct is `( field: value, ... )`, a list is `[ ... ]`, and enum variants are written by name, `Sampler(Builtin(Kick))`, just as in the code. JSON would work too, but JSON has no enums, so the same channel would read `{"Sampler": {"Builtin": "Kick"}}`. RON is easier to read and to fix by hand, which matters for a program you learn from: you can open a song in a text editor and see your own data structures.

serde: describe the data once

Rust’s standard way to turn values into text (or bytes) and back is the **serde** crate, short for *serialize* and *deserialize*. `serde` does not know any format itself. It defines two traits, `Serialize` and `Deserialize`, and format crates such as `ron` or `serde_json` use them.

You almost never write those traits by hand. A **derive macro** writes them for you:

```
use serde::{Deserialize, Serialize};

#[derive(Debug, Clone, PartialEq, Serialize, Deserialize)]
pub struct Channel {
    pub name: String,
    pub kind: ChannelKind,
    pub volume: f32,
    pub muted: bool,
    #[serde(default, skip_serializing_if = "Option::is_none")]
    pub program: Option<u8>,
    #[serde(default, skip_serializing_if = "is_centred")]
    pub pan: f32,
    // ...the reverb send, the MIDI channel a drum is sent on, and the
    // channel's effects, each left out at its default.
}
```

The macro reads the struct definition at compile time and generates code that writes each field by name, and reads them back. For this to work, every field’s type must be

Serialize too, so ChannelKind, SampleSource, Drum, NoteEvent, Scale, PitchClass, Track, Clip and the rest derive it as well. String, numbers, bool, Vec, Option and PathBuf come with serde.

The derive is turned on by asking for serde's derive feature in Cargo.toml. **Features** are optional parts of a crate that you switch on when you need them:

```
serde = { version = "1", features = ["derive"] }
```

The `#[serde(default, skip_serializing_if = ...)]` lines are how the file format grows quietly. A field that is at its default is not written at all, and a field missing from an older file is filled in with its default. That is why the drum kit above, which uses nothing beyond the first layout, is written exactly as the first version of the format wrote it: a channel in the middle with all of it in the reverb says nothing about pan or reverb send.

Keeping the rules while reading: from and into

A Tempo cannot be made outside 10 to 522 BPM, because its field is private and `Tempo::new` clamps it. A derived `Deserialize` would set the field directly from the file, so a file saying `tempo: 9000.0` would create a tempo the rest of the program promises can never exist.

The fix is to tell serde to go through a type we control:

```
#[derive(Debug, Clone, Copy, PartialEq, PartialOrd, Serialize,
Deserialize)]
#[serde(from = "f32", into = "f32")]
pub struct Tempo {
    bpm: f32,
}

impl From<f32> for Tempo {
    fn from(bpm: f32) -> Self {
        Tempo::new(bpm)
    }
}

impl From<Tempo> for f32 {
    fn from(tempo: Tempo) -> Self {
        tempo.bpm
    }
}
```

`#[serde(from = "f32")]` means “read an `f32`, then convert it with `From<f32>`”. That conversion is `Tempo::new`, which clamps. The test `values_out_of_range_are_clamped` edits a saved file to say 9000 BPM and checks that it loads as 522. `into = "f32"` does the reverse when saving, which also makes the file say `tempo: 130.0` instead of `tempo: (bpm: 130.0)`. `Note` does the same with its MIDI number.

Always in letters and English

The app can show note names in letters (C, D, E) or in solfège (Do, Re, Mi), and its texts in English or Spanish (see [Two languages](#)). A file must not depend on either: a song saved by someone who reads solfège must open for someone who reads letters. So `to_text` and `from_text` set both for as long as they run:

```
let file = Language::English.with(|| NoteNaming::Letters.with(||
ProjectFile::from(project)));
let mut text = NoteNaming::Letters.with(||
ron::ser::to_string_pretty(&file, style));
```

`with` sets the naming for the thread, runs the closure, and puts the old naming back, so the window draws in solfège again on its next frame.

A separate type for the file

`Project` itself does not derive `serde`. Its fields are private because they must agree with each other (one row per channel, clips that point at real patterns), and a derived `Deserialize` would fill them straight from the file, agreeing or not. So the file has its own struct:

```

#[derive(Debug, Serialize, Deserialize)]
struct ProjectFile {
    format: u32,
    #[serde(default, skip_serializing_if = "crate::Mixer::is_default")]
    mixer: crate::Mixer,
    tempo: Tempo,
    scale: Scale,
    #[serde(default, skip_serializing_if = "TimeSignature::is_common")]
    time_signature: TimeSignature,
    channels: Vec<Channel>,
    patterns: Vec<PatternFile>,
    tracks: Vec<Track>,
    clips: Vec<Clip>,
    // ...sections, the loop region, time, tempo and key markers, loose
    // timing, swing, the chord sheet, the band, where each part came
    // from, and the locked channels: each left out when it is empty.
}

#[derive(Debug, Serialize, Deserialize)]
struct PatternFile {
    name: String,
    #[serde(default, skip_serializing_if = "Option::is_none")]
    color: Option<usize>,
    #[serde(default, skip_serializing_if = "Option::is_none")]
    time: Option<TimeSignature>,
    #[serde(default, skip_serializing_if = "Option::is_none")]
    player: Option<Player>,
    length: usize,
    /// One list of notes per channel, in the order of `channels`.
    notes: Vec<Vec<NoteEvent>>,
}

```

A `ProjectFile` accepts anything shaped right. Turning it into a `Project` is where the checking happens, and it is written as a `TryFrom` implementation, the standard trait for “a conversion that can fail”:

```

impl TryFrom<ProjectFile> for Project {
    type Error = FileError;

    fn try_from(file: ProjectFile) -> Result<Self, Self::Error> {
        let mut project = Project::empty();
        // ...checks on the mixer, the time signature and each channel's
        // mix, effects and MIDI channel...
        for channel in file.channels {
            project.add_channel(channel);
        }
        for (index, saved) in file.patterns.into_iter().enumerate() {
            // `Project::empty` already has one pattern; add the others.
            if index > 0 {
                project.add_pattern();
            }
            let Some(pattern) = project.pattern_mut(index) else { ... };
            pattern.set_length(saved.length);
            for (channel, notes) in saved.notes.into_iter().enumerate() {
                for note in notes {
                    if pattern.add_note(channel, note).is_none() {
                        return invalid(/* where, and what is wrong */);
                    }
                }
            }
        }
        // ...tracks, clips, sections, markers, the chord sheet and the
        band...
        Ok(project)
    }
}

```

The project is rebuilt **through its front door**: `add_channel`, `add_pattern`, `add_note`, `add_clip`, the same methods the app calls when you click. Those methods already keep every rule (see [The project](#)), so a file can never create a project that the window could not have created. When a method refuses (a note that does not fit, a clip that plays a pattern that is not there), the error says where, so a person editing the file by hand can fix it: “the project file is damaged: clip 1 (pattern 99, track 1) does not fit in the playlist”.

`type Error = FileError;` is an **associated type**: part of the trait that each implementation chooses. With `TryFrom` in place, the standard library also provides the matching `try_into`, so `let project: Project = file.try_into()?;` would work too.

Saving goes the other way with `From<&Project> for ProjectFile`, which copies the data out with `to_vec()` and `map`. It cannot fail, so it is a `From`, not a `TryFrom`.

The version number

```
pub const FORMAT_VERSION: u32 = 29;
```

`FORMAT_VERSION` is the newest layout this program can read. A file does not always use it: `format_for` writes the **lowest** format that can hold the project. A project that uses nothing new stays format 1, so an older Ferrous Lineup can still open it. The starting song needs format 9, because its synth channel plays a General MIDI program; a song with a chord sheet needs at least format 12; one with a horn section or a channel effect needs 28, and a band playing another style's drums 29. The long comment above `FORMAT_VERSION` says, for every number, what it added and what an older program would get wrong without it; [Project file format](#) has the same list as a table.

A file from a **newer** version must be refused with a clear message, not read wrongly. The obvious way is to read a tiny struct first:

```
#[derive(Deserialize)]
struct Header {
    format: u32,
}
```

By default, `serde` skips fields that a struct does not list, so a `Header` can be read from the whole file. But to skip the rest of the file, RON has to look ahead through all of it, number by number. On a long song such as Boléro, that took over a minute, even in a release build. So the whole file is read first, and only if that fails does `format_number` read the number from the first few bytes:

```

pub fn from_text(text: &str) -> Result<Project, FileError> {
    // A text editor may start the file with a byte-order mark, which is
    // not part of the RON text or of the format number.
    let text = text.strip_prefix('\u{feff}').unwrap_or(text);
    let file: ProjectFile = match NoteNaming::Letters.with(||
ron::from_str(text)) {
        Ok(file) => file,
        Err(err) => {
            return Err(match format_number(text) {
                Some(format) if format > FORMAT_VERSION =>
FileError::NewerFormat(format),
                _ => err.into(),
            });
        }
    };
    if file.format > FORMAT_VERSION {
        return Err(FileError::NewerFormat(file.format));
    }
    NoteNaming::Letters.with(|| Project::try_from(file))
}

/// The format number at the start of a project file, `(format: 25, ...`,
/// read without the RON reader.
fn format_number(text: &str) -> Option<u32> {
    let rest = text.trim_start().strip_prefix('(')?.trim_start();
    let rest = rest.strip_prefix("format")?.trim_start();
    let rest = rest.strip_prefix(':')?.trim_start();
    let digits = rest
        .find(|c: char| !c.is_ascii_digit())
        .unwrap_or(rest.len());
    rest[..digits].parse().ok()
}

```

`to_text` always writes `format` first, so this looks only where the number must be. A newer file often fails to read at all, because of fields or names this version does not know; then the format number alone says why. If the file is so broken that even the number cannot be read, the original RON error is the one reported. And if the file parses but still says a newer format, it is refused too, because the same text may mean something else in that format. A byte-order mark at the start (some text editors add one) is stripped first, so it cannot hide the `(`.

The test `files_from_the_future_are_refused` reads a file one format newer than this version, `(format: 30, sparkles: true)` today, and expects `NewerFormat(30)`, and the message “this file uses format 30, made by a newer version of Ferrous Lineup (this one reads up to format 29)”.

Errors as an enum

```
#[derive(Debug)]
pub enum FileError {
    /// The file could not be read or written: missing, no permission,
    /// disk full...
    Io(io::Error),
    /// The text is not a valid project file. The error says where.
    Parse(ron::error::SpannedError),
    /// The project could not be turned into text (should never happen).
    Write(ron::Error),
    /// The file was saved by a newer version of Ferrous Lineup.
    NewerFormat(u32),
    /// The file is shaped right, but what it says does not make sense.
    Invalid(String),
}
```

This is the same pattern as `EngineError` (see [A custom error type](#)) and `SampleError`: one variant per kind of problem, a `Display` implementation with a message for people (in the app’s language), and `std::error::Error` so it works with any Rust error handling. Two more things make it pleasant to use.

From for ?. Inside a function that returns `Result<_, FileError>`, the `?` operator can only pass an error on if it knows how to turn it into a `FileError`. These implementations teach it:

```
impl From<io::Error> for FileError {
    fn from(err: io::Error) -> Self {
        FileError::Io(err)
    }
}
```

With one for `io::Error`, one for `ron`’s parse error and one for its write error, a function can mix calls from both libraries and use `?` on all of them:

```
pub fn load(path: &Path) -> Result<Project, FileError> {
    let text = fs::read_to_string(path)?; // io::Error -> FileError::Io
    from_text(&text)
}
```

source. The `Error` trait has an optional method that returns the lower-level error a value wraps. Error-reporting tools follow it to print a chain like “could not load: no such file”. `FileError::Io` returns its `io::Error`; `Invalid` has nothing underneath and returns `None`.

```
fn source(&self) -> Option<&(dyn std::error::Error + 'static)> {
    match self {
        FileError::Io(err) => Some(err),
        FileError::Parse(err) => Some(err),
        FileError::Write(err) => Some(err),
        FileError::NewerFormat(_) | FileError::Invalid(_) => None,
    }
}
```

`&(dyn Error + 'static)` is a **trait object**: a reference to *some* value that implements `Error`, without saying which type. That lets one method return an `io::Error` in one case and a `ron` error in another.

Saving without losing the old file

```
pub fn save(project: &Project, path: &Path) -> Result<(), FileError> {
    let text = to_text(project)?;
    let temporary = path.with_extension(format!("{EXTENSION}.tmp"));
    if let Err(err) = fs::write(&temporary, text) {
        let _ = fs::remove_file(&temporary);
        return Err(err.into());
    }
    if let Err(err) = fs::rename(&temporary, path) {
        let _ = fs::remove_file(&temporary);
        return Err(err.into());
    }
    Ok(())
}
```

Writing straight over `song.lineup` has a nasty failure: if the program crashes or the disk fills up halfway, the file is half old, half new, and neither version can be loaded. Writing to `song.lineup.tmp` first and then renaming it over the original avoids that, because the operating system replaces a file by renaming in a single step. Either the rename happened and the new file is complete, or it did not and the old file is untouched. On Windows, Rust's `fs::rename` also replaces an existing file, so the same code works everywhere. The WAV and video exports and the recovery copy are written the same way.

`to_text` always ends lines with `\n` (RON would use `\r\n` on Windows), so a song is the same file on every system, and a project saved on one computer shows no difference in version control when saved on another.

The tests save to `std::env::temp_dir()`, the system's folder for temporary files (`/tmp` on Linux, something under your user folder on Windows), with the process id in

the name so two test runs at once do not collide, and delete the file afterwards.

Damaged files never crash the app

A project file can be cut short by a full disk, edited by hand, or be something else under a `.lineup` name. MIDI files come from every program there is. Reading a file must end in a project or in an error message, never in a crash. The test file `damaged_files.rs` checks this the way a **fuzzer** does: it damages good files in many small ways and reads each result.

The damage comes from a tiny random number generator, xorshift, written in the test:

```
struct Damage(u64);

impl Damage {
    fn next(&mut self) -> u64 {
        self.0 ^= self.0 << 13;
        self.0 ^= self.0 >> 7;
        self.0 ^= self.0 << 17;
        self.0
    }
}
```

It starts from a fixed seed, so every run damages the files the same way, and a failure shows up again on the next run. It needs no new dependency. For project files and chord sheets, it replaces numbers with extreme ones (`-1` , `65535` , `18446744073709551615` , `NaN`), cuts pieces out, and repeats pieces. For MIDI files, it changes, cuts and repeats bytes. Each test reads 600 damaged copies. A project that reads must also save and read back.

The test catches a panic with `std::panic::catch_unwind` , so a failure prints the damaged file that caused it, up to its first 4,000 characters:

```
fn must_not_panic<T: std::fmt::Debug>(input: T, read: impl FnOnce(&T)) {
    if panic::catch_unwind(AssertUnwindSafe(|| read(&input))).is_err() {
        let shown = format!("{input:?}");
        let shown: String = shown.chars().take(4000).collect();
        panic!("reading this damaged file panicked:\n{shown}");
    }
}
```

Overflow is the kind of mistake it catches. A file whose key change sat at step `18446744073709551615`, the largest `usize` , fails validation as it should, but writing

the error message computed `marker.start + 1` to say “step ...”, and that overflowed. Rust treats overflow differently in the two build profiles. In a **debug** build, `+` panics on overflow (“attempt to add with overflow”). In a **release** build it wraps around to 0, silently. So a run in release mode over 15,000 damaged files saw nothing, and the same files in debug mode crashed. The messages use `saturating_add(1)`, which stops at the largest number instead of overflowing, and CI runs the tests in debug mode (see [How the code is tested](#)).

In the app: the File menu

The app side is in `crates/lineup-app/src/app/files.rs`. Its first lines explain an unusual layout:

```
//! This is a child module of `app` (its file is `app/files.rs`), so it
can
//! use the private fields of [LineupApp]: in Rust, private means
"visible
//! in this module and the modules inside it".
```

`app.rs` declares `mod files;`, and Rust looks for that module in `app/files.rs`. The file holds another `impl LineupApp` block. A type can have any number of `impl` blocks, in any modules of its crate, so a big type can be split by topic. Being a child of `app`, `files` sees the private fields `self.project`, `self.audio` and so on, which a sibling module like `playlist` could not.

Choosing a file

File dialogs come from the **rfd** crate (“Rusty File Dialog”), which shows the system’s own dialog: GTK or KDE on Linux (through the desktop portal, or the `zenity` program), the Finder’s panel on macOS, the Explorer dialog on Windows.

```
fn project_dialog() -> rfd::FileDialog {
    rfd::FileDialog::new().add_filter(tr("Ferrous Lineup project"), &
[file::EXTENSION])
}

if let Some(path) = project_dialog().pick_file() {
    self.open_now(path);
}
```

`pick_file` waits until the user picks a file or cancels, and returns an `Option<PathBuf>`. The window does not redraw while the dialog is open, but the music keeps playing, because it runs on the audio thread. On a Linux system with neither a portal nor `zenity`, `pick_file` returns `None` straight away; the dialog simply does not appear.

Save As gives the file a `.lineup` extension even if you type another one, so `song.txt` is saved as `song.lineup`, not as something that another importer would try to open.

“Save changes?”

How does the app know there are unsaved changes? It keeps a copy of the project as it was last opened or saved, and compares. But `Project` derives `PartialEq`, so `!=` walks every channel, note and clip. That is cheap once, but a drag edits the project on every frame, and comparing the whole song every frame is wasted work. `FileState` therefore caches the answer:

```
/// Whether the project differs from `saved`, as last worked out.
dirty: bool,
/// Set when the project may differ from `saved`; `dirty` is worked
/// out again the next time it is asked for.
check_dirty: bool,
```

An edit sets `check_dirty`, but only if the project was not already dirty. While it is dirty, more frames of a drag do not ask for the whole comparison again:

```
pub(super) fn project_edited(&mut self) {
    if !self.dirty {
        self.check_dirty = true;
    }
}
```

A change made in one go (a click), an undo, a redo, and the end of a drag can all make the project match the saved copy again, so they always ask for a new comparison with `project_reverted` (see [Telling the audio thread what changed](#)). The cached answer is read every frame for the title, but the comparison itself runs only when the flag is set:

```

pub(super) fn has_unsaved_changes(&mut self) -> bool {
    if self.file.check_dirty {
        self.file.dirty = match &self.file.saved {
            Some(saved) => self.project != *saved,
            None => true, // recovered work has never been saved
        };
        self.file.check_dirty = false;
    }
    self.file.dirty
}

```

So a step clicked on and then off again clears the star, and so does a note dragged away and back, once you let go.

Anything that would throw the project away waits for an answer first:

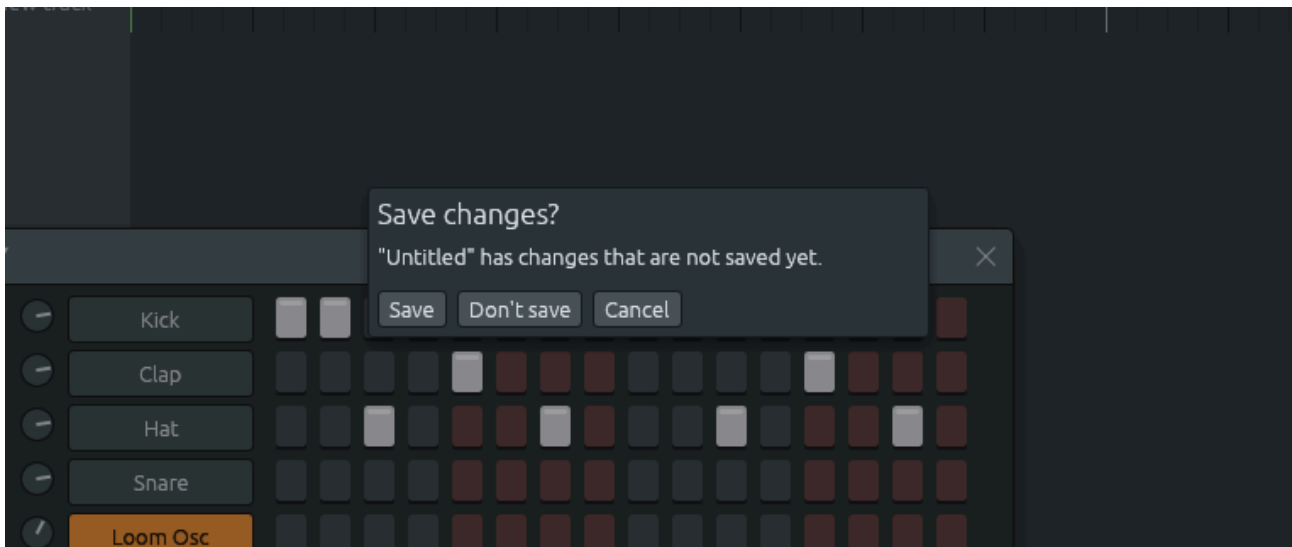
```

pub(super) enum Pending {
    /// A chord sheet with a band.
    New,
    /// An empty pattern and a drum kit.
    NewBeat,
    Open,
    OpenPath(PathBuf),
    Demo(Demo),
    Import,
    ImportMidi,
    ImportChordSheet,
    Quit,
}

pub(super) fn request(&mut self, ctx: &egui::Context, action: Pending) {
    // ...a quit during an export asks about the export first...
    if self.has_unsaved_changes() || ... {
        self.file.pending = Some(action);
    } else {
        self.perform(ctx, action);
    }
}

```

While `pending` holds something, `unsaved_changes_dialog` shows an `egui::Modal`, a window that dims everything behind it and takes all the clicks:



Save saves and then carries out the waiting action, unless saving failed or the user cancelled the Save As dialog; Don't save carries it out straight away; Cancel, Escape or a click outside forgets it. The `pending` value is the whole state machine: `None` means "nothing waiting", `Some(Pending::Open)` means "the user wants to open a file, once they decide about the changes".

`pub(super)` makes the enum visible to the parent module (`app`), which needs it for `FileState`, but no further.

The style editor has its own unsaved draft. Quitting with an edited draft asks **Save style changes?** as well, so saving the song never silently throws the style away.

Closing the window

Clicking the window's `×` asks the operating system to close it. egui reports the request, and the app can say no:

```
let close_requested = ctx.input(|input|
input.viewport().close_requested());
if close_requested
    && !self.file.quitting
    && (self.has_unsaved_changes() ||
self.style_editor.has_unsaved_changes())
{
    ctx.send_viewport_cmd(egui::ViewportCommand::CancelClose);
    self.file.pending = Some(Pending::Quit);
}
```

When the user then picks Save or Don't save, `perform(Pending::Quit)` sets `quitting` and sends `ViewportCommand::Close`, which is let through this time.

Quitting while a WAV or video export is still running asks **An export is still running**, with **Stop it and quit** and **Keep exporting**. Stopping waits until the export has stopped and removed its unfinished file, and then closes, so a half-written file is never left behind looking finished.

Shortcuts

```
const SAVE: KeyboardShortcut = KeyboardShortcut::new(Modifiers::COMMAND,
Key::S);
const SAVE_AS: KeyboardShortcut =
    KeyboardShortcut::new(Modifiers::COMMAND.plus(Modifiers::SHIFT),
Key::S);

let pressed = |shortcut| ctx.input_mut(|input|
input.consume_shortcut(&shortcut));
if pressed(SAVE_AS) {
    self.save_as();
} else if pressed(SAVE) {
    self.save();
}
```

- `Modifiers::COMMAND` is Ctrl on Linux and Windows and ⌘ on macOS, so each system gets the shortcut its users expect. The File menu shows the right one, too: “Ctrl+S” or “⌘S”.
- `consume_shortcut` checks the keys and removes them from the input, so a text field does not also see them.
- Ctrl+S would also match Ctrl+Shift+S (the extra Shift is ignored), so the longer shortcut is checked first.

These `const` values call `KeyboardShortcut::new` while compiling. That is allowed because egui marked it as a `const fn`, a function simple enough to run at compile time.

The title bar

The window title shows the song’s name and a star for unsaved changes, “song.lineup* · Ferrous Lineup”. A demo shows its own name and says it is read-only, “Für Elise (demo, read-only) · Ferrous Lineup”: demos are built into the app, so Save asks for a new file, as Save As does. The app sends a `ViewportCommand::Title` only when the text changes, not on every frame.

Opening a file from outside

```
let mut args = std::env::args_os().skip(1);
let first = args.next();
```

`ferrous-lineup song.lineup` starts with the song open, which is also what a file manager does when you double-click a file associated with the app. (The same first argument can instead be `--trace`, `--demos` or `--help`, which do their job without opening a window; see [The playback trace](#).) `args_os` returns the arguments as `OsStrings`, exactly as the system gave them. `std::env::args` would panic on a file name that is not valid UTF-8, which Linux allows. `skip(1)` skips the first argument, which is the program's own name.

A `.lineup` file dropped on the window is opened too, after the same “Save changes?” question, and a MIDI file, a chord sheet or a recording is imported. Other dropped WAV files go to the channel rack (see [Importing recordings](#) for which WAV is which).

File > Open recent

File > Open recent lists the last ten projects opened or saved, most recent first. The list is kept in egui's stored data, so it survives a restart. Each item shows the file's name, with the full path in its tooltip. Opening a file that no longer loads removes it from the list instead of leaving a dead entry.

The list holds paths, not copies of projects. If a project is moved or renamed outside the app, the old path stays until an attempt to open it fails.

A copy of unsaved work

The app writes a recovery copy of the project while it has unsaved changes. The first change writes one at once, so the first edit of a session is not the one a crash loses; after that it writes every thirty seconds:

```
/// How often unsaved work is written while it keeps changing.
const INTERVAL: Duration = Duration::from_secs(30);
```

A window left alone draws no frames, and so would never get round to writing. So when a copy is due, `autosave` asks egui for a frame at that moment with

`request_repaint_after` : the copy catches up within about half a minute of a change, even if nobody touches the app.

The files are in a `recovery` folder inside the app's data folder, the one `eframe::storage_dir(APP_NAME)` gives it. On Linux that is usually `~/.local/share/ferrouslineup/recovery` . Each run of the app names its files after itself, such as `4211-1793047912612-0.lineup` and `4211-1793047912612-0.info` : the process number, the time it started in milliseconds, and a count. Why each run needs its own is in [Two windows at once](#) below. The copy is **not** written next to the project, because a project may be on a read-only medium, and an autosave file in a folder of songs would surprise anyone looking at it. The small `.info` file carries the project's path and name, so a recovered song comes back as itself, not as "Untitled".

The copy is written by `file::save` , which writes a temporary file and renames it. A crash in the middle leaves the previous copy whole; the info file is written after the project, so a crash between them loses the name rather than giving the project the wrong one.

At startup, if a copy is there, a modal asks **Recover unsaved work?** and names the song:

- **Recover it** puts the project back, with its path and name, and removes the copy. It also loads the file on disk, if it still exists, as the "saved" version, so the recovered work correctly shows the unsaved star; undoing back to the file clears it. If the file is gone, the recovered project is simply unsaved.
- **Discard it** removes the copy and leaves the project the app would otherwise have opened.

A clean exit removes the copy, because there is nothing to recover. The copy that remains is the sign of a crash, a kill, or a power cut. A copy whose project will not load is removed at startup rather than offered. A single `recovery.lineup` straight in the data folder, as older versions wrote it, is offered too.

Two windows at once

Nothing stops you from starting Ferrous Lineup twice. If every run wrote the same copy, the second window would find the first one's copy at startup and offer it as if the first had crashed, while it was still open; and each window's autosave would replace the other's copy.

So each run writes its own copy, and next to it a third file, `<run>.lock` , which it **locks** before the first copy is written and holds while it runs. A file lock is a promise kept by the operating system: only one open file can hold it, and when the program that holds

it ends, the system lets go of it, even in a crash or a power cut. That makes it the right question to ask at startup: “is the run that wrote this copy still going?” If its lock can be taken, it is not, and the copy is offered; if not, the copy belongs to a window that is still open and is left alone.

The standard library has had file locks since Rust 1.89 (`File::lock`, `File::try_lock` and friends). `try_lock` does not wait: it says at once whether it got the lock, and its error tells “someone else holds it” apart from “this system cannot lock files”:

```
fn take_lock(path: &Path) -> Lock {
    // A copy whose lock file is gone was left by a run that closed with
    // work unsaved: the file is made again, to be taken.
    let Ok(file) = OpenOptions::new()
        .create(true)
        .truncate(false)
        .write(true)
        .open(path)
    else {
        return Lock::Unknown;
    };
    match file.try_lock() {
        Ok(()) => Lock::Taken(file),
        Err(TryLockError::WouldBlock) => Lock::Held,
        Err(TryLockError::Error(_)) => Lock::Unknown,
    }
}
```

The lock lasts as long as the `File` value: dropping it lets go. So a window that finds a copy keeps the `File` in the `Recovered` value until you answer the question; a third window opened meanwhile cannot take the lock and does not offer the same copy twice. When the app closes, the `Drop` of `Recovery` removes its lock file; a copy of unsaved work stays, without its lock, to be offered next time. A system that cannot lock files (`Lock::Unknown`) offers the copy anyway: offering it twice loses nothing, never offering it would.

If two runs crashed, there are two copies. The newest is offered first, and the next waits until the work recovered before it is saved or let go, so recovering one never replaces another.

Drawing the windows with egui: layout, zoom, themes, icons

Files: `crates/lineup-app/src/main.rs`, `crates/lineup-app/src/app.rs`,
`crates/lineup-app/src/theme.rs`, `crates/lineup-app/src/widgets.rs`,
`crates/lineup-app/src/icons.rs`, `crates/lineup-app/src/app/view.rs`,
`crates/lineup-app/src/app/shortcuts.rs` and `crates/lineup-app/src/app/about.rs`

The window is drawn with [egui](#), a GUI library written in Rust, and [eframe](#), egui's framework for running it as a native app on Linux, macOS and Windows. This chapter is about how the main window is put together each frame, the look, how rows of controls fit any window size, the zooms, the windows that float over the playlist, and the Help menu. The editors themselves, the channel rack, the piano roll and the playlist, have the [next chapter](#).

Immediate mode

Most GUI toolkits are **retained mode**: you create button objects once, keep them in a tree and attach callbacks to them. egui is **immediate mode**: on every frame, you call a function that describes the *whole* UI from your own data, and the library draws it and tells you what was clicked.

```
if ui.button("▶ Play").clicked() {  
    self.set_playing(true);  
}
```

That line both draws the button and checks whether it was clicked. There are no button objects to keep in sync with your state. The state is just the fields of your struct, and the UI is always a picture of it. This fits Rust nicely, because there are no callbacks holding references to your data.

The price is that a frame cannot go back. Something drawn early in the frame cannot know what is drawn later in it. Several pieces of this chapter (the hint bar, wrapping groups, zoom) solve that the same way: they use what the *previous* frame found, a sixtieth of a second old.

main.rs

```
fn main() -> eframe::Result {
    // The first command-line argument, if there is one, is a file to
    open,
    // or what to do instead of opening the window.
    let mut args = std::env::args_os().skip(1);
    let first = args.next();
    match first.as_ref().and_then(|arg| arg.to_str()) {
        Some("--trace") => std::process::exit(trace(args)),
        Some("--demos") => { /* print the demos' names */ return Ok(()); }
        Some("--help" | "-h") => { /* print how to use it */ return
Ok(()); }
        _ => {}
    }
    let path = first.map(PathBuf::from);

    let mut viewport = eframe::egui::ViewportBuilder::default()
        .with_title(app::APP_NAME)
        .with_inner_size([1280.0, 800.0])
        .with_min_inner_size([800.0, 520.0])
        .with_icon(window_icon());
    // ...inside a Flatpak, the window takes the sandbox's app id...

    let options = eframe::NativeOptions {
        viewport,
        ..Default::default()
    };

    eframe::run_native(
        app::APP_NAME,
        options,
        Box::new(|cc| Ok(Box::new(app::LineupApp::new(cc, path)))),
    )
}
```

- `main` returns a `Result`. If it returns `Err`, Rust prints the error and exits with a failure code.
- `ViewportBuilder::default().with_title(...)` is the **builder pattern**: each method takes the builder and returns it modified, so calls chain.
- `..Default::default()` is **struct update syntax**: “every field I did not list takes its default value”.
- `Box::new(|cc| ...)` passes a closure that `eframe` calls once to create our app. `Box` puts a value on the **heap**. It is needed here because `eframe` accepts *any* type implementing its `App` trait, and to store a value whose exact type it does not know, it needs a pointer to it: `Box<dyn App>`. The `dyn` keyword means “some type implementing this trait, decided at run time” (a **trait object**).
- `"--help" | "-h"` is an **or-pattern**: the arm matches either string.

LineupApp: the state

`LineupApp` holds everything the window needs: the project, the undo history, the open windows and their own state, the file, the settings, and the connection to the audio thread. That connection is an enum:

```
enum Audio {  
    /// Boxed: the engine is much bigger than a `String`, and every  
    /// `Audio` would otherwise take as much room as the biggest variant.  
    Running(Box<AudioEngine>),  
    Unavailable(String),  
}
```

It captures a real situation: either there is a running engine, or there is the reason there is not. You cannot have both, and you cannot have neither, so the type cannot represent an impossible state. Compare that with the usual alternative, an `Option<AudioEngine>` plus an `Option<String>`, which allows four combinations of which two make no sense.

An enum is as big as its biggest variant, plus room to say which one it is. `Box` keeps the big engine on the heap, so `Audio` is the size of a pointer or a `String`, whichever is bigger. Clippy points this out with its `large_enum_variant` lint.

Using it:

```
fn send(&mut self, command: Command) {  
    if let Audio::Running(engine) = &mut self.audio {  
        engine.send(command);  
    }  
}
```

`if let` is a one-arm `match`: if `self.audio` is `Running`, bind the engine inside to `engine` and run the block. Matching on `&mut self.audio` borrows the engine mutably instead of moving it out of the struct.

If the computer has no working sound card, the app still opens and the status bar says why in orange: “Audio unavailable: ...”. A failure becomes data we display, rather than a crash.

One frame

`eframe` calls `ui` every time the window needs repainting:


```
impl eframe::App for LineupApp {
    fn ui(&mut self, ui: &mut egui::Ui, _frame: &mut eframe::Frame) {
        if let Audio::Running(engine) = &mut self.audio {
            engine.collect_garbage();
        }
        self.check_trace();
        let ctx = ui.ctx().clone();
        self.before_drawing(&ctx);
        self.panels(ui, &ctx);
        let playing = self.song_position().filter(|_| self.playing);
        let area = self.main_area(ui, &ctx, playing);

        // The channel rack and the piano roll float on top of the
        playlist.
        let playhead = self.pattern_playhead();
        self.bring_window_to_front(&ctx);
        self.rack_window(&ctx, area, playhead);
        self.piano_roll_window(&ctx, area, playhead);
        // The chord sheet is drawn last, so it opens over the others.
        self.chord_sheet_window(&ctx, area, playing);
        // ...ear training, the lead sheet, the mixer, the shortcuts,
        // About, the style editor, Settings and the dialogs...
        self.unsaved_changes_dialog(&ctx);
        self.update_title(&ctx);
        self.end_frame_for_undo(&ctx);
    }

    fn on_exit(&mut self) {
        self.forget_recovery();
    }
}
```

The order is the design:

1. **collect_garbage** frees what the audio thread sent back (see [The trash queue](#)).
2. **before_drawing** does everything that must happen before anything is drawn: it notes whether a click is closing a menu, sets the language and the note names for the frame, closes an undo step on a mouse press, writes the recovery copy when due, reads the keyboard shortcuts and the zoom keys, reads MIDI input, and checks on exports and imports running in the background. While playing it asks egui for another frame at once (`ctx.request_repaint()`), so the playhead moves; otherwise egui draws only when something happens.
3. **panels** draws the bars along the edges: the menus with the hint bar, the toolbar, the status bar, and the harmony keys. Panels must come before the central area, which takes whatever space is left.
4. **main_area** draws the playlist (or the chord sheet, when it fills the main window) and returns the rectangle it took.
5. **The windows** float over that rectangle. Windows drawn later are on top, so the chord sheet comes last of the three editors.

6. `end_frame_for_undo` decides whether the undo step ends (see [Undo and redo](#)).

`on_exit` runs once when the app closes cleanly: there is no unsaved work to recover, so this run's recovery copy goes.

The many `self.something(ui)` methods keep `ui` short. Each closure passed to a panel can borrow `self` mutably, because each one runs and finishes before the next starts, which the borrow checker can see.

A theme is one function

egui draws every widget from a `Style`: colours in `style.visuals`, sizes and gaps in `style.spacing`. Nothing is painted with a hard-coded colour unless the code asks for one. So the whole look is one function, `theme::apply`, called once when the app starts:

```
pub fn apply(ctx: &egui::Context) {
    ctx.set_theme(egui::Theme::Dark);
    ctx.style_mut_of(egui::Theme::Dark, |style| {
        let visuals = &mut style.visuals;
        visuals.panel_fill = PANEL;
        visuals.window_fill = WINDOW;
        visuals.selection.bg_fill = ACCENT; // selected and switched-on
things
        visuals.selection.stroke = Stroke::new(1.0, ON_ACCENT);
        // ...shadows, corners, warning and error colours...
        let widgets = &mut visuals.widgets;
        for (state, fill) in [
            (&mut widgets.inactive, BUTTON),
            (&mut widgets.hovered, BUTTON_HOVER),
            (&mut widgets.active, BUTTON_PRESSED),
            (&mut widgets.open, BUTTON_HOVER),
        ] {
            state.bg_fill = fill;
            state.weak_bg_fill = fill;
            state.bg_stroke = Stroke::new(1.0, OUTLINE);
            // ...
        }
        // ...spacing...
    });
}
```

The look borrows from pattern-based studios: dark blue-grey panels, raised grey buttons, an orange light for whatever is selected or switched on, and black “LCD” displays for numbers such as the tempo. Three things to notice in the code:

- `style_mut_of(Theme::Dark, ...)` changes the *dark* style only. egui keeps one style for dark and one for light, and the app always uses dark.
- The `for` loop walks an **array of pairs of mutable references**. Each `&mut` `widgets.inactive` borrows a *different field*, so Rust allows several of them at once: the borrow checker tracks fields of a struct separately.
- The colours are `const` items: `Color32::from_rgb` is a `const fn`, so it runs at compile time and costs nothing when the app runs.

Views still have their own constants for the things only they draw (the playlist's rows, the piano roll's keys, the step buttons), with values that match the theme.

Changing the style for a few widgets

The tempo display is a `DragValue`, egui's number you drag or type, but drawn black with green-blue digits. Its style is changed only inside its frame:

```
pub fn lcd_frame(
    ui: &mut Ui,
    label: &str,
    width: f32,
    add: impl FnOnce(&mut Ui) -> Response,
) -> Response {
    wrapping(ui, |ui| {
        egui::Frame::new()
            .fill(LCD)
            // ...
            .show(ui, |ui| {
                // ...the label above, written smaller if it is long...
                ui.vertical(|ui| {
                    ui.label(label);
                    let visuals = &mut ui.visuals_mut().widgets;
                    for state in [&mut visuals.inactive, &mut
visuals.hovered, &mut visuals.active] {
                        state.bg_fill = LCD;
                        state.fg_stroke.color = LCD_TEXT;
                        // ...
                    }
                    ui.style_mut().override_font_id =
Some(FontId::monospace(16.0));
                    add(ui)
                })
                .inner
            })
        .inner
    })
}
```

Every `ui` has its own copy of the style (an `Arc<Style>`; `style_mut` makes a private copy the first time it is changed). The frame's contents get a child `ui`, so the change stays inside the frame and the rest of the toolbar is untouched. `add` is a closure that *returns* the widget's `Response`, so the caller can still ask `changed()` outside:

```
let tempo = theme::lcd_frame(ui, tr("TEMPO"), 62.0, |ui| {
    ui.add(egui::DragValue::new(&mut bpm).range(Tempo::MIN..=Tempo::MAX))
});
```

`Tempo`'s field is private, so the value is copied into a local `bpm`, the widget edits that, and a new `Tempo` is built if it changed. The `Tempo` stays valid whatever is typed.

`Tempo::MIN..=Tempo::MAX` is an **inclusive range**: `a..b` leaves `b` out, `a..=b` keeps it.

A label longer than the display, as some are in Spanish, is written a little smaller to fit, so the toolbar keeps its width in every language.

The hint bar: a message from the future

Next to the menus, a line says what the control under the mouse does. That line is drawn *first*, at the top of the window, but the widgets that know what the mouse is on are drawn *later* in the same frame. In immediate mode there is no going back to repaint the bar.

So the bar shows what was found on the **previous** frame. Widgets leave their hint in egui's memory, `ctx.data`, which is a map from an `Id` to a value of any type:

```
pub fn hint(response: Response, text: impl Into<String>) -> Response {
    let text = text.into();
    if response.hovered() {
        response
            .ctx
            .data_mut(|data| data.insert_temp(hint_id(), text.clone()));
    }
    response.on_hover_text(text)
}
```

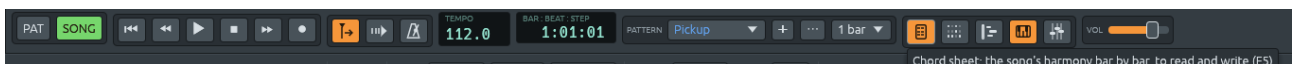
At the start of the next frame, the bar takes it out, so it does not linger once the mouse moves away:

```
pub fn start_frame(&mut self, ctx: &egui::Context) {
    let text = ctx.data_mut(|data| data.remove_temp::(hint_id()));
    if text != self.text {
        // The mouse moved onto or off a widget: draw once more, so the
        // bar does not lag a frame behind until the next event.
        ctx.request_repaint();
    }
    self.text = text;
}
```

`remove_temp::` uses the **turbofish** `::<String>` to say which type to take out: the map can hold values of many types under the same `Id`. The `request_repaint` is the subtle part. `egui` only draws a frame when something happens. When the mouse moves onto a button, that frame records the hint, but the bar still shows the old one. Asking for one more frame shows the new hint straight away, instead of whenever the mouse next moves.

Rows that fit any window

The toolbar holds the play mode, the transport, the tempo and position displays, the pattern selector, the five window buttons, the count-in, the metronome and the master volume. In a window 1280 points wide it fits on one row; a test, `the_toolbar_fits_on_one_row_at_1280_points`, checks that it stays so. A narrower or zoomed window must still reach every control.



`egui` lays widgets out one after another. `ui.horizontal` puts them in a row and keeps going past the edge if they do not fit: whatever is past the edge is still there, just not drawn. `ui.horizontal_wrapped` is the same row, except that a widget that would not fit starts a new row. The toolbar and the harmony keys bar are wrapping rows, so on a narrow window the second part of them moves down.

Groups that wrap: last frame's width

The toolbar's buttons are in groups, each inside a dark frame. A wrapping row decides to wrap by asking a widget how big it wants to be before placing it. A frame cannot answer that: `Frame::show` draws its contents into whatever room is left, and only knows its size afterwards. So `theme::wrapping` uses the frame before. `egui` redraws the window many times a second, so last time's width is a very good guess:

```

let id = ui.next_auto_id().with("wrapping");
let last_width = ui.ctx().data(|data| data.get_temp::

```

- `ui.next_auto_id()` is the id egui would give the next widget. It depends on how many widgets came before in this `ui`, so it is the same every frame as long as the toolbar is drawn in the same order. That makes it a key for remembering something about one group.
- `get_temp:: asks egui's memory for an f32; the turbofish tells Rust which type to look for, because nothing else in the line says so.`
- The `if let ... && ... chain` (a "let chain") only runs `end_row` when there is a width from last time, the layout wraps, the row already has something in it, and the group does not fit. The very first frame has no width yet, so the group may stick out for one frame, which nobody sees.

`theme::group` and `lcd_frame` wrap their frames in it, and `keep_together` offers the same to any group of widgets, with a `guess` for the very first frame. That first frame matters inside a window: egui windows grow to fit their contents, and **a single frame of contents that is too wide is enough to widen a window for good**. The piano roll's toolbar keeps its groups together this way, so a long scale name (F# major's "F# G# A# B C# D# E#") moves its group to a second row instead of widening the window.

Text that gets shorter, and right to left

Wrapping suits buttons. For a long line of text, like a status message, it is often better to cut it short. Every text in egui has a *wrap mode*: `Wrap` breaks it onto more lines, `Truncate` cuts it with "...", and `Extend` lets it run on. A whole area can set it for all its text through its style:

```

ui.style_mut().wrap_mode = Some(egui::TextWrapMode::Truncate);

```

A truncated label shows the whole text when you point at it, so nothing is lost.

The status bar has two parts: messages on the left, and the playlist's zoom controls on the right. If the messages were drawn first, a long one ("Audio unavailable: could not read the device settings...") would take all the room and push the zoom out. So the bar is laid out *right to left*: the zoom goes in first, from the right edge, and the messages get whatever is left:

```
fn status_bar(&self, ui: &mut egui::Ui) {
    ui.with_layout(egui::Layout::right_to_left(egui::Align::Center), |ui|
    {
        view::zoom_control(ui);
        ui.separator();
        ui.with_layout(egui::Layout::left_to_right(egui::Align::Center),
        |ui| {
            ui.style_mut().wrap_mode = Some(egui::TextWrapMode::Truncate);
            self.status_messages(ui);
        });
    });
}
```

In a right-to-left layout the first widget is the rightmost one, which is why `zoom_control` adds its + button before the percentage and the – button.

Zoom

There are two kinds of zoom.

The **interface size** makes everything bigger or smaller: text, buttons, menus. egui draws in *points*, not pixels. How many pixels a point takes is `pixels_per_point`: the screen's own scale (1.0, 1.5 or 2.0 on most screens) times egui's *zoom factor*. `ctx.set_zoom_factor` changes the second.

The **areas** zoom what they draw without changing the text. As in most music programs, the playlist and the piano roll zoom their two directions separately: horizontal zoom makes bars and steps wider (less time on screen, in more detail), vertical zoom makes tracks and piano keys taller. The channel rack's steps are square buttons, so it has one zoom for both.

Pointer over	Ctrl and + / – (or wheel)	Alt and + / – (or wheel)	Ctrl+0
Playlist	bars wider / narrower	tracks taller / shorter	both to 100%
Piano roll	steps wider / narrower	keys taller / shorter	both to 100%

Pointer over	Ctrl and + / – (or wheel)	Alt and + / – (or wheel)	Ctrl+0
Channel rack	bigger / smaller	the same	100%
Anything else, the chord sheet too	interface bigger / smaller	the same	100%

What is under the pointer?

The keys and the wheel zoom what the pointer is over. In Rust, “one of these kinds” is an `enum`, and a variant can carry a value:

```
pub(crate) enum Target {
    Area(Area), // Area::Main (the playlist), Area::Rack, Area::PianoRoll
    Interface,
}
```

`Target::at` finds it. `egui` keeps a stack of *layers*: the background where the panels are drawn, then each window above it, then menus and tooltips. `ctx.layer_id_at(pos)` says which layer is on top at a point:

```
fn at(ctx: &egui::Context, pos: egui::Pos2, playlist: egui::Rect) ->
Target {
    match ctx.layer_id_at(pos) {
        Some(layer) if layer.id == super::rack_id() =>
Target::Area(Area::Rack),
        Some(layer) if layer.id == PianoRoll::id() =>
Target::Area(Area::PianoRoll),
        Some(layer) if layer.order == egui::Order::Background &&
playlist.contains(pos) => {
            Target::Area(Area::Main)
        }
        _ => Target::Interface,
    }
}
```

Each arm has a **match guard**, the `if` after the pattern. The pattern `Some(layer)` matches any layer, and the arm is only taken if the guard is true too; otherwise `match` tries the next arm. The last arm, `_`, catches everything left.

The toolbars, the harmony keys and the playlist are all drawn on the same background layer, so the layer alone cannot tell them apart. The playlist therefore remembers where it was drawn, and the guard checks `playlist.contains(pos)`. The zoom keys are read at the start of a frame, before anything is drawn, so this is the

rectangle from the previous frame. If the pointer is outside the window, the keys zoom what was clicked last.

The chord sheet is text, so it follows the interface size, in its own window or in the playlist's place. In the playlist's place it covers the playlist, and the playlist's last rectangle would be a lie, so `zoom_shortcuts` passes an empty one instead:

```
let playlist = match self.chord_sheet.fills_window() {
    true => egui::Rect::NOTHING,
    false => self.playlist.area(),
};
```

`Rect::NOTHING` contains no point, so the guard is never true, and the keys fall through to `Target::Interface`.

Every way of zooming (keys, wheel, touchpad pinch) calls the same few methods on `Target`, and only they look at which kind it is:

```
fn step(self, ctx: &egui::Context, axis: Axis, bigger: bool) {
    match self {
        Target::Area(area) => area.step_zoom(ctx, axis, bigger),
        Target::Interface => {
            ctx.set_zoom_factor(interface_step(ctx.zoom_factor(),
bigger));
        }
    }
}
```

`Target::Area(area)` both checks the variant and takes the `Area` out of it into the variable `area`. The interface has a single size, so `axis` is not used there: Ctrl and Alt do the same.

A zoom per area and per axis

Each area keeps one number per direction, in egui's memory, under a key made of both:

```
fn key(self, axis: Axis) -> egui::Id {
    let axis = if self.two_way() { axis } else { Axis::Horizontal };
    egui::Id::unique(("zoom-2", self, axis))
}
```

The rack is not `two_way`, so both its axes share the horizontal key: zooming it “vertically” zooms it both ways. The values are stored with `insert_persisted`, so egui saves them with the window positions and they are the same the next time the app

starts. The `"zoom-2"` name is there because an older meaning of “zoom” (a scale for everything in the window, text included) was saved under `"zoom"`: a new name means old values are simply not found, instead of being read the wrong way.

Every way of zooming in steps goes through `step_zoom`, which walks a ladder of sizes:

```
const STEPS: [f32; 19] = [
    0.25, 0.33, 0.4, 0.5, 0.6, 0.75, 0.9, 1.0, 1.1, 1.25, 1.5, 1.75, 2.0,
    2.5, 3.0, 4.0, 5.0, 6.0,
    8.0,
];
```

Each step is roughly a fifth bigger than the one before, so it takes as many steps to go from 100% to 200% as from 400% to 800%. That is how zoom feels natural: our eyes compare sizes by ratio, not by difference. Each area and axis has its own range within the ladder (tracks and keys can only get so short before their names no longer fit), and the buttons are greyed out at the ends. The interface sizes are a shorter list of their own, 50% to 250%, the same ones Settings > Appearance offers.

Reading the wheel ourselves

egui gives a single `zoom_delta()` for Ctrl+wheel, with no idea of direction. So the app reads the frame’s raw wheel events and sorts them by the modifier keys held:

```
for event in &input.raw.events {
    if let egui::Event::MouseWheel { unit, delta, modifiers, .. } = event
    {
        // Up or right zooms in. Shift turns a wheel sideways.
        let amount = if delta.y != 0.0 { delta.y } else { delta.x };
        if modifiers.alt {
            tall.add(*unit, amount);
        } else if modifiers.command {
            wide.add(*unit, amount);
        }
    }
}
```

`if let` with a pattern is how Rust picks one variant out of an enum and names its fields at the same time; `..` ignores the fields we do not need.

On Linux (X11), one notch of the wheel sometimes arrives as two events, and zooming two steps per notch feels jumpy. So the notches are added up and zoom at most one step per frame:

```
fn apply(self, target: Target, ctx: &egui::Context, axis: Axis) {
    if self.notches != 0.0 {
        target.step(ctx, axis, self.notches > 0.0);
    }
    if self.smooth != 1.0 {
        target.scale(ctx, axis, self.smooth);
    }
}
```

A frame lasts a few milliseconds, so turning the wheel quickly still zooms step by step. A touchpad sends many tiny movements instead of notches (`MouseWheelUnit::Point`); those multiply the zoom smoothly.

Zooming into what you point at

When the zoom changes, the step and the note under the pointer must stay under the pointer. The grid is inside a scroll area, so this means choosing a new scroll offset. The thing under the pointer is at `from + anchor` in the contents, where `from` is how far the view is scrolled and `anchor` is where the pointer is in the view. Zooming by `ratio` moves it to `(from + anchor) * ratio`. For it to stay under the pointer, the new scroll must be that minus `anchor`. In the piano roll:

```
let x = KEY_WIDTH + (from.x + anchor.x - KEY_WIDTH) * ratio.x - anchor.x;
let y = (from.y + anchor.y) * ratio.y - anchor.y;
self.scroll_to = Some(vec2(x.max(0.0), y.max(0.0)));
```

The `KEY_WIDTH` terms are there because the piano keyboard at the left does not stretch: only what is right of it grows.

One more thing had to be turned off. `egui`'s scroll areas have a **rubber band**: scroll past the end, and the view springs back. On the frame of a zoom, the scroll area still knows the *old* size of the contents, so the new offset looks like scrolling past the end, and the view would slide back after each zoom. `widgets::no_rubber_band` turns it off for the grids:

```
pub fn no_rubber_band(ui: &mut egui::Ui) {
    ui.spacing_mut().scroll.kinetic.rubber_band = false;
}
```



The channel rack is drawn with ordinary egui widgets, which have no zoom of their own. `widgets::zoom_contents` makes them bigger by multiplying the text sizes and the spacing of the window's style, as `lcd_frame` changes its colours: only inside the rack.

Windows over the playlist

The channel rack, the piano roll, the chord sheet, the mixer and the others are `egui::Window`s, which float over everything. They are kept inside the playlist's rectangle with `Window::constrain_to(area)`, so a window never hides the toolbar, the harmony keys or the status bar:

```
let playlist = egui::CentralPanel::default_margins().show(ui, |ui| { /*
... */ });
let area = playlist.response.rect;
```

Every `show` in egui returns an `InnerResponse`: `.inner` is what the closure returned (here the playlist's actions) and `.response` tells where it was drawn.

Which windows a song opens on

What you study in a song is its bars and chords; what you edit in a beat is the steps of a pattern. So the app decides what to show by what was opened:

```
/// What a song opens on: the chord sheet in front, since its bars and
/// chords are what you study and write in a song, and the rack out of
/// the way (F6 brings it back).
fn show_chord_sheet(&mut self) {
    self.chord_sheet.open = true;
    self.rack_open = false;
    self.bring_to_front = Some(chord_sheet::window_id());
}

/// What a beat opens on: the rack in front, and no chord sheet.
fn show_beat(&mut self) {
    self.chord_sheet.open = false;
    self.show_rack();
}
```

Song demos, File > New and an imported chord sheet call `show_chord_sheet`; File > New beat calls `show_beat`. Opening a saved file leaves the windows as they are, since you chose them yourself. A demo is a song or a style, an `enum`, and only the songs open on the sheet:

```
if let Demo::Song(_) = demo {
    self.show_chord_sheet();
}
```

The app itself opens on the windows that were open when it was last closed.

`app/windows.rs` keeps them as one small struct of `bool`s, one for each window of the View menu:

```
#[derive(Debug, Clone, Copy, PartialEq, Serialize, Deserialize)]
#[serde(default)]
pub(super) struct OpenWindows {
    chord_sheet: bool,
    lead_sheet: bool,
    rack: bool,
    piano_roll: bool,
    harmony_keys: bool,
    /// The Patterns and Library panel.
    browser: bool,
    mixer: bool,
    ear_training: bool,
}
```

Its `Default` is the first time: the chord sheet, the harmony keys and the Library over the playlist, with no rack. `#[serde(default)]` on the struct fills a window added to the list later with its value in `Default`, so an older save still reads. The style editor and the dialogs are left out: each opens for something, a style or a question, that the next run does not have.

`App::save`, which `eframe` calls every 30 seconds and when the window closes, writes the struct among the preferences kept by name (see [Kept by name, not by type](#)), and `LineupApp::new` reads it back:

```
fn save(&mut self, storage: &mut dyn eframe::Storage) {
    self.preferences.put(windows::NAME, &self.open_windows());
    self.preferences.save(storage);
}
```

```
let windows: OpenWindows = preferences::get(ctx,
NAME).unwrap_or_default();
```

Writing them only on save, rather than at every click on a window button, keeps the many places that open and close windows free of that chore.

`the_app_first_opens_on_the_chord_sheet_then_with_the_windows_left_open` checks both runs: the first opens on the sheet, and after closing it with F5 and opening the mixer with F9, the next start finds the mixer open and the sheet closed.

Windows the size of what they show

egui remembers each window's size, and saves it with the rest of its memory when the app closes. That is right for a window you work in, like the piano roll: you made it big because you wanted it big. It is wrong for a window that is as big as what it shows: drag the mixer wide once, and it would open across the whole playlist ever after, with its channels in one corner. So there are two kinds of window:

```
pub enum Opening {
    /// Each time it opens: a window as big as what it shows.
    EveryTime,
    /// The first time it opens after the app starts: a window to work
    /// in, which keeps the size you give it while the app runs.
    FirstTime,
}
```

The mixer, the shortcuts list, ear training, the twelve-tone row, the pitch-class sets and the micromodes open **every time** at the size of their contents, up to the height of the playlist, and scroll past that. The piano roll, the chord sheet, the lead sheet and

Settings open at their own size the **first time** after the app starts, and keep a size you give them while it runs.

`widgets::open_at` does it. It finds out whether the window is opening now (it was not drawn in the last frame), and if so gives it a fixed size for that one frame:

```
let first = ctx.data_mut(|data| {
    !std::mem::replace(data.get_temp_mut_or(id.with("has-opened"), false),
true)
});
if opens(ctx, id) && (first || opening == Opening::EveryTime) {
    ctx.request_discard("a window opens at its own size");
    window.fixed_size(size)
} else {
    window
}
```

Three things in it are worth a look:

- `std::mem::replace(place, new)` puts `new` in `place` and returns what was there. Here it reads “had this window opened before?” and marks it as opened, in one step. The flag is stored with `get_temp_mut_or`, which egui does not save, so it is `false` again each time the app starts.
- egui lays a window out as it draws it, so on its first frame the window is drawn before anything knows how big its contents are. `request_discard` asks egui to throw that frame away and draw it again; you only see the second one, already at the right size.
- The “size” given is a narrow width and the greatest height. egui grows a window to fit wider contents but never makes it taller than they are, so the window ends up exactly as big as what it shows. For that to work, nothing in it may fill all the room it is given, such as a scroll area that does not shrink to its contents.

A window whose contents change while it is open, like the mixer when a channel is added or the rack when another pattern has fewer channels, is fitted again with `fit_when_changed`. It hashes a summary of the contents (each channel’s name and instrument, say) and fits the window again when the hash changes.

Clicks that do what you meant

Two helpers in `widgets.rs` make the grids react to the click you meant.

A quick tap. On a touchpad, a tap can press and release the button within one frame. egui’s `press_origin()` is forgotten as soon as the button is up, so on that frame it is

`None`. `press_pos` reads the press from the frame's list of events instead, which also says *where* the button went down, not where the pointer is at the end of the frame:

```
pub fn press_pos(
    input: &egui::InputState,
    only: Option<egui::PointerButton>,
) -> Option<egui::Pos2> {
    input.events.iter().find_map(|event| match *event {
        egui::Event::PointerButton { pos, button, pressed: true, .. }
            if only.is_none_or(|only| only == button) => Some(pos),
        _ => None,
    })
}
```

`find_map` walks the events until the closure returns `Some`, and returns that.

`Option::is_none_or` reads well here: “no button was asked for, or it is this one”.

A click that closes a menu. Open a drop-down list, then click on the grid to close it: `egui` closes the menu, but the click also reaches the grid, which would draw a note, place a clip or turn on a step. In most programs that click only closes the menu. So at the start of each frame, before anything is drawn and while a menu that will close is still open, the app remembers whether the button now down went down while a menu was open:

```
pub fn track_dismissing(ctx: &egui::Context) {
    let id = dismissing_id();
    let (pressed, down, released) = ctx.input(|input| {
        (
            press_pos(input, None).is_some(),
            input.pointer.any_down(),
            input.pointer.any_released(),
        )
    });
    let open = egui::Popup::is_any_open(ctx);
    ctx.data_mut(|data| {
        let dismissing = data.get_temp_mut_or(id, false);
        if pressed {
            *dismissing = open;
        } else if !down && !released {
            *dismissing = false;
        }
    });
}
```

The grids ask `dismissing(ctx)` and ignore the press, the drag and the release that belong to it. The value lives in `egui`'s temporary memory, a handy place for a little state shared between widgets that do not know each other.

Remembering between runs

egui keeps some state of its own between frames in its *memory*: the zoom factor, where each window is, how wide a resizable panel is. eframe can save that memory when the app closes and load it when it starts. It does so only with its `persistence` feature, which is off by default because it brings in `serde` and `ron`. Turning it on is one line in the workspace's `Cargo.toml`:

```
eframe = { version = "0.37", features = ["persistence"] }
```

With it, eframe also remembers the size and position of the main window. It writes a file called `app.ron` in the place each system keeps app data (on Linux, `~/.local/share/ferrouslineup/`), every 30 seconds and when the window closes.

The app keeps most of its own lasting choices in the same memory, with `insert_persisted` instead of `insert_temp`: the area zooms, the settings, the recent files, the library's filters. Each module stores its own under an `Id` of its own, and eframe saves them all together. Anything kept with `insert_temp` is forgotten when the app closes.

Kept by name, not by type

egui finds a value in its memory by two things: the `Id` it was stored under and its Rust type, through the type's `TypeId`. A `TypeId` is a number the compiler works out from the type's path, its fields and the crate it is in, and the crate's part changes with the crate's version and with the version of the compiler. So after an update of the app, or a build with a newer Rust, or a field added to the type, `get_persisted` looks under a new number and finds nothing: the setting is back to its default, and the old value sits unread in `app.ron`. The SoundFont showed it: with the same `app.ron`, version 0.4.0 played from GeneralUser GS and a build numbered 0.4.1 had it off.

eframe has a storage of its own, the same `app.ron`, which finds values by a name: a `&str` that stays the same whatever the version. eframe hands it to the app only twice: as `cc.storage` when the app starts, and to `App::save`, every 30 seconds and when the window closes. The module `app/preferences.rs` bridges the two. It keeps every preference by name, written as RON text, in one map:

```
/// Each preference by name, written as RON. egui's context holds one
/// handle to the map and the app another, to save it.
#[derive(Clone, Default)]
pub(super) struct Preferences(Arc<Mutex<BTreeMap<String, String>>>);
```

A handle to the map lives in egui's temporary memory, so any code with the `Context` reaches it, as with any other value kept there:

```
/// The preference kept as `name`, if there is one and it can be read.
pub(super) fn get<T: DeserializeOwned>(ctx: &egui::Context, name: &str) ->
Option<T> {
    let text = Preferences::of(ctx).map().get(name).cloned()?;
    ron::from_str(&text).ok()
}

/// Keeps `value` as the preference `name`.
pub(super) fn set<T: Serialize>(ctx: &egui::Context, name: &str, value:
&T) {
    if let Ok(text) = ron::to_string(value) {
        Preferences::of(ctx).map().insert(name.to_string(), text);
    }
}
```

The app reads the map from eframe's storage before anything else is made,

```
if let Some(storage) = cc.storage {
    preferences::Preferences::of(&cc.egui_ctx).load(storage);
}
```

keeps another handle to it, and gives it back to eframe to write:

```
fn save(&mut self, storage: &mut dyn eframe::Storage) {
    self.preferences.save(storage);
}
```

The `Arc<Mutex<...>>` lets both handles share one map. The lock is held only while a line is read or written, and only the window's thread takes it. Because the values are RON text until they are asked for, a type can change its fields and still read what an older version kept, with `#[serde(default)]` for fields that were not there yet.

The sound settings (`SoundFonts`) are kept this way, as `"sound-font"`, and so are the open windows, as `"open-windows"`. The other choices above are still in egui's memory, so an update can forget them; moving one over is a matter of calling `preferences::get` and `preferences::set` with a name instead of `get_persisted` and `insert_persisted` with an `Id`.

Cargo features

A *feature* is a switch a crate offers to turn on optional code. Inside eframe the code that saves memory sits behind `#[cfg(feature = "persistence")]`, so without the

feature it is not even compiled. Turning on a feature in one crate can turn on features in others: `eframe`'s `persistence` turns on `egui`'s, which adds `serde` derives to the types `egui` keeps in its memory. You can see what a crate offers in the `[features]` table of its `Cargo.toml`, or on its page on `docs.rs`.

Pictures painted, not typed

The toolbar's five window buttons and the count-in and metronome toggles are square buttons with a picture; the name, what the window shows and its key appear when the mouse is over one. An emoji would be the easy way to draw such a picture, but an emoji is a character, drawn by a font, and some systems have no font with those characters: there the buttons would show empty boxes. So the pictures in `icons.rs` are **painted**, with lines, rectangles and polygons, the same way on every system.

Each picture is a plain function with the same signature, and Rust can name that signature as a type:

```
/// A picture: drawn by a function of the painter, the square it fills
/// and the colour to draw it in.
pub type Picture = fn(&Painter, Rect, Color32);
```

`fn(&Painter, Rect, Color32)` is a **function pointer** type. Any function with that signature can be stored in a variable of type `Picture`, put in an array, or passed to another function, without a `Box` or a generic.

Each picture is drawn in a square, and its points are given as fractions of the square, from 0 to 1, so the same picture can be drawn at any size:

```

/// The point `x`, `y` of the way across and down `rect`, each from 0 to
1.
fn at(rect: Rect, x: f32, y: f32) -> Pos2 {
    rect.min + vec2(x * rect.width(), y * rect.height())
}

/// The mixer: three faders, each at its own level.
pub fn mixer(painter: &Painter, rect: Rect, color: Color32) {
    for (x, level) in [(0.2, 0.62), (0.5, 0.3), (0.8, 0.5)] {
        line(painter, rect, (x, 0.0), (x, 1.0),
color.gamma_multiply(0.6));
        let knob = Rect::from_center_size(at(rect, x, level), vec2(6.0,
3.5));
        painter.rect_filled(knob, 1.0, color);
    }
}

```

The chord sheet is a page with chord names two to a line, the rack three rows of steps with some lit, the piano roll a keyboard with three notes, the harmony keys a keyboard with two black keys, the metronome its body and arm.

The toolbar draws its five window buttons from an array. Each entry holds the name, the `bool` that says whether that window is open, the picture and the hint:

```

let windows: [(&str, &mut bool, icons::Picture, &str); 5] = [
    ("Chord sheet", &mut self.chord_sheet.open, icons::chord_sheet,
    "the song's harmony bar by bar, to read and write (F5)"),
    ("Channel rack", &mut self.rack_open, icons::rack,
    "the steps of the selected pattern (F6)"),
    // ... the piano roll, the harmony keys, the mixer
];
for (name, open, picture, about) in windows {
    let button = icons::toggle(ui, name, *open, picture);
    // ...the hint, and a click opens or closes the window...
}

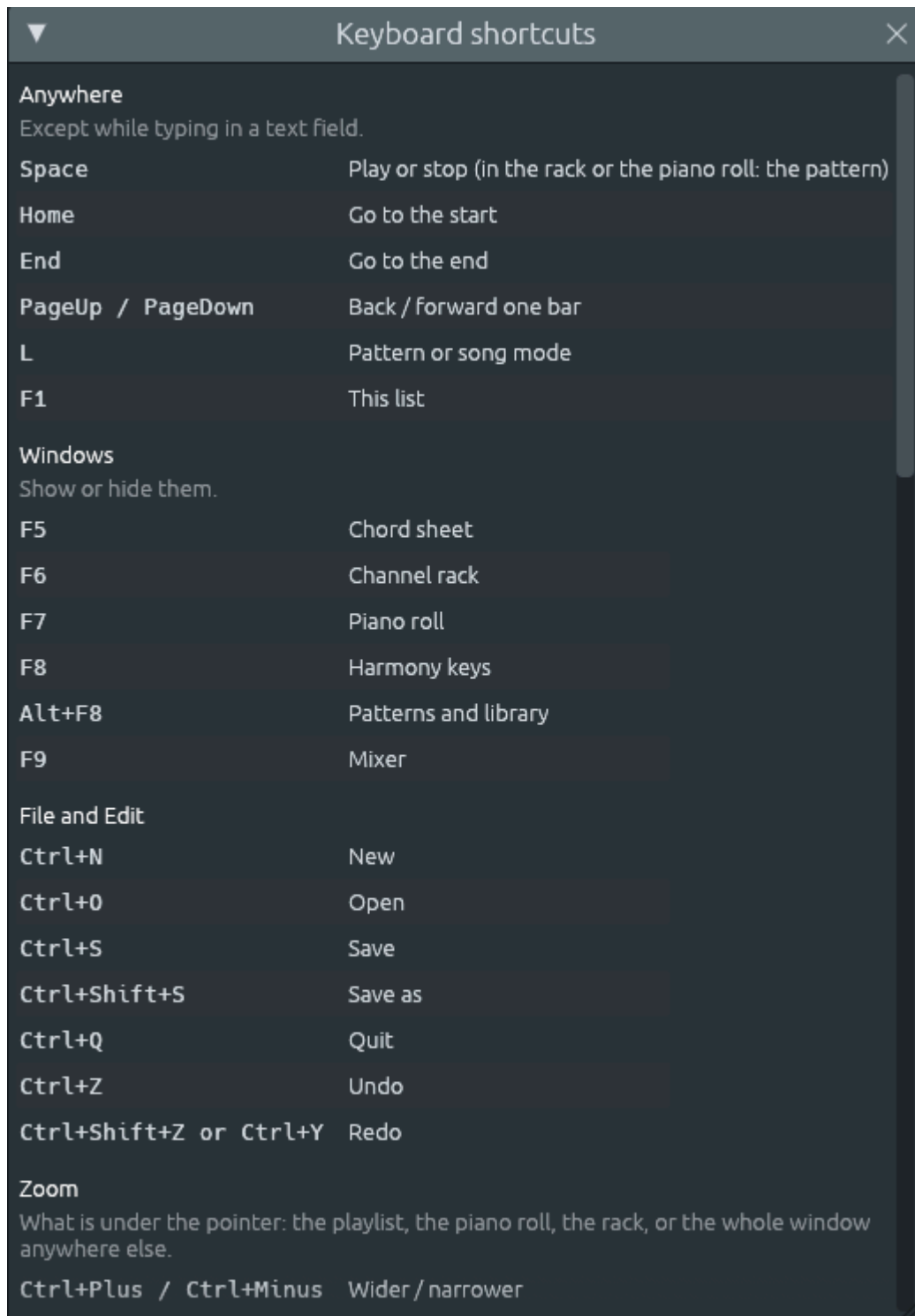
```

There are five `&mut` borrows of `self` in one array. The borrow checker allows it because each one borrows a **different field**: `self.chord_sheet.open`, `self.rack_open`, `self.piano_roll.open` and so on. Two `&mut` to the same field would not compile.

The test `every_picture_stays_in_its_square` paints each picture into a 16-point square and checks that no shape goes outside it, and `the_window_buttons_open_and_close_their_windows` clicks each of the five and checks that its window opens, that the button lights, and that a second click closes it.

Every key in a list

Help > Keyboard shortcuts, or **F1**, opens a window with every key the app answers to, section by section.



The list is data: a `Vec` of sections, each with a title, a note on when its keys work and its lines. A line's keys are one of four kinds:

```
enum Keys {
    /// Any of these does it.
    Any(Vec<KeyboardShortcut>),
    /// The first does the first half of what the line says, the second
the
    /// second: "Up / down".
    Pair(KeyboardShortcut, KeyboardShortcut),
    /// The mouse wheel, turned while holding these.
    Wheel(Modifiers),
    /// Keys that are not one shortcut, written out.
    Written(&'static str),
}
```

Wherever the app has a constant for a shortcut, the list uses that same constant:

`files::SAVE`, `edit::UNDO`, `piano_roll::OCTAVE_UP`, `view::MIXER`. So if a key changes, the list changes with it. Turning a `Keys` into text uses `egui's format_shortcut`, which writes “Ctrl+Shift+Z” on Linux and Windows and “⌘⇧Z” on a Mac, the way the menus write them.

Because the list is data, it can also be tested as data.

`no_two_global_shortcuts_share_a_key` collects the keys of the sections that work anywhere and fails if any key is listed twice, since that key would do two things at once.

Help: the manual and About

Help > User manual opens the user manual’s web pages in the app’s language. The address lives in a function of `about.rs`, so the Help menu and the About window both ask it:

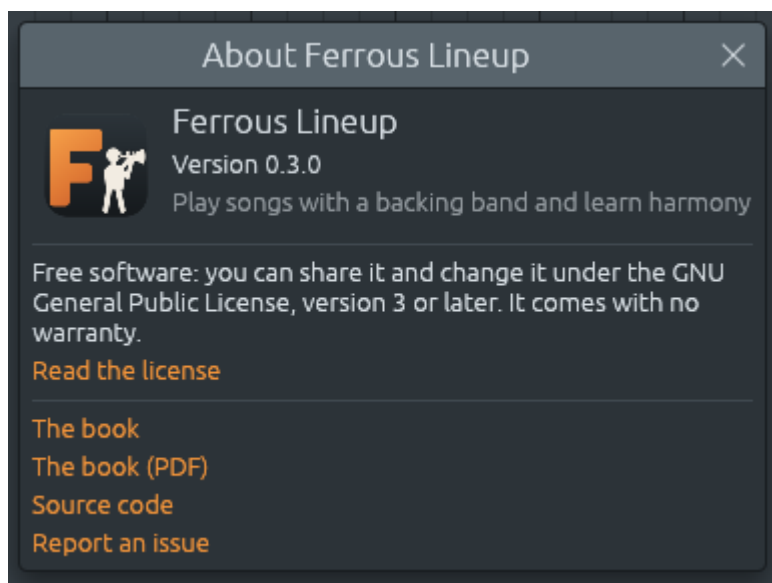
```
pub(super) fn manual() -> &'static str {
    match i18n::current_language() {
        Language::English => MANUAL,
        Language::Spanish => MANUAL_ES,
    }
}
```

The `match` has no `_` arm on purpose. Add a third language, and the compiler stops here until you decide which manual its readers get. `pub(super)` makes `manual` visible to `app.rs` and to the modules inside it, such as `shortcuts.rs`, where the Help menu is, and to nothing else.

In the menu, the button asks `egui` to open that address in a new tab of the web browser:

```
if ui.button(crate::i18n::tr("User manual")).clicked() {
    ui.ctx()
        .open_url(egui::OpenUrl::new_tab(super::about::manual()));
    ui.close();
}
```

Help > About Ferrous Lineup says which version you are running, under which license, and where to find the user manual, this book (each also as a PDF), the code and the list of known problems. It is the first thing to look at before reporting one.



Everything it says about the program comes from `Cargo.toml`, read while the program is compiled:

```
const ISSUES: &str = concat!(env!("CARGO_PKG_REPOSITORY"), "/issues");

ui.label(format!("{}", tr("Version"), env!("CARGO_PKG_VERSION")));
ui.weak(tr(env!("CARGO_PKG_DESCRIPTION")));
```

- `env!("NAME")` is a macro that the compiler replaces with the value of the environment variable `NAME` *while it compiles*, as a `&'static str`. Cargo sets `CARGO_PKG_VERSION`, `CARGO_PKG_DESCRIPTION`, `CARGO_PKG_REPOSITORY` and a few more from the `[package]` section of `Cargo.toml` (the app takes its version, license and repository from the workspace's `[workspace.package]`). A variable that is not set is a compile error, not a surprise for the user.
- `concat!` joins literals into one at compile time, so `ISSUES` is a constant too. `format!` could not be used there: it builds a `String` while the program runs, and a `const` must be known before it runs.

So when a release changes the version in `Cargo.toml`, the window changes with it, and nobody has to remember a second place. The [release builds](#) compare the tag

with this same number.

The picture is the one the desktop shows for the app's window: `WINDOW_ICON`, 256 × 256 pixels of red, green, blue and alpha bytes, built into the program with `include_bytes!`. egui draws pictures from **textures**, images kept on the graphics card, so the bytes are made into a texture the first time the window opens and kept in egui's memory:

```
fn icon(ctx: &egui::Context) -> egui::TextureHandle {
    let id = egui::Id::unique("about-icon");
    if let Some(texture) = ctx.data(|data| data.get_temp(id)) {
        return texture;
    }
    let size = crate::WINDOW_ICON_SIZE as usize;
    let image = egui::ColorImage::from_rgba_unmultiplied([size, size],
crate::WINDOW_ICON);
    let texture = ctx.load_texture("about-icon", image,
egui::TextureOptions::LINEAR);
    ctx.data_mut(|data| data.insert_temp(id, texture.clone()));
    texture
}
```

A `TextureHandle` works like an `Arc`: a clone is cheap, and the texture stays on the graphics card while any clone of it is alive. The one kept in egui's memory means the picture is sent to the card once, not on every frame. `WINDOW_ICON` is a constant of `main.rs`, not marked `pub`; `about.rs` can still read it as `crate::WINDOW_ICON`, because a private item is visible to its own module and to every module inside it.

`ui.hyperlink_to(text, url)` draws a link. Clicking it does not start the browser: egui adds an `OutputCommand::OpenUrl` to what the frame gives back, and eframe asks the system to open the address. That makes the click easy to test.

`help_about_shows_the_version_the_license_and_where_the_book_is` opens **Help > About**, checks that the version line and every link are drawn, clicks **The book**, and checks that the frame asked for the book's address, without any browser opening. `help_user_manual_opens_the_manual_in_the_apps_language` does the same for **Help > User manual**, in English and in Spanish.

Testing a window without a screen

All the tests named in this chapter draw the app without a screen.

`egui::Context::run_ui` (or `run_pass`) takes the input of one frame, the mouse position, the keys and the clicks, and returns everything egui would hand to the graphics card, without drawing it. A test can then find a widget by its `Id` or its text,

check where it was placed, click it in the next frame, and read the app's state. `app/ux_tests.rs` holds most of them; [How the code is tested](#) has the rest of the story.

One detail of these tests is worth knowing. What `run_ui` returns includes the font's texture. egui checks, only in debug builds, that such a texture is not dropped without being used, and panics if it is. So tests that keep the output clear it first:

```
// The font texture is not drawn anywhere here.  
output.textures_delta.clear();
```

A test that passes in a release build and fails in a debug build is usually one of these checks: `debug_assert!` is compiled only into debug builds, and a library can have many of them. That is why CI runs the tests in a debug build.

The editors: channel rack, piano roll and playlist

Files: `crates/lineup-app/src/channel_rack.rs`, `crates/lineup-app/src/piano_roll.rs`, `crates/lineup-app/src/playlist.rs`, `crates/lineup-app/src/widgets.rs` and `crates/lineup-app/src/app.rs`

Three windows edit the music itself. The **playlist** fills the main window: patterns placed as clips on tracks, which together make the song. The **channel rack** (F6) shows one pattern as rows of steps, one row per channel. The **piano roll** (F7) shows one channel of one pattern as notes on a grid, with high notes at the top. The rack and the piano roll float over the playlist, as the [previous chapter](#) describes.

All three are built the same way, and that is the first thing this chapter is about.

Drawing returns actions

None of the editors changes the project. Each one only looks at it and returns what the user did, as a list of actions. The rack's are an enum:

```

pub enum RackAction {
    PlayPattern,
    // ...
    SelectChannel(Option<usize>),
    ToggleStep { channel: usize, step: usize },
    ClearStep { channel: usize, step: usize },
    Preview(usize),
    SetVolume { channel: usize, volume: f32 },
    ToggleMute(usize),
    Remove(usize),
    OpenPianoRoll(usize),
    AddDrum(Drum),
    AddSynth,
    // ...sending over MIDI, instruments...
    LoadSample { channel: Option<usize>, path: PathBuf },
}

pub fn show(
    ui: &mut Ui,
    project: &Project,
    pattern: usize,
    selected: Option<usize>,
    playhead: Option<usize>,
    playback: Option<RackPlayback>,
    locks: bool,
) -> Vec<RackAction>

```

`project: &Project` is a **shared borrow**: the function may read the project but cannot change it, and the compiler enforces that. The app then applies the actions:

```

let rack_actions = rack
    .show(ctx, |ui| {
        // ...
        channel Rack::show(ui, &self.project, self.pattern, selected,
playhead, /* ... */)
    })
    .and_then(|window| window.inner)
    .unwrap_or_default();
for action in rack_actions {
    self.apply(action);
}

```

`Window::show` returns `None` when the window is closed, and its `inner` is `None` when the window is folded up, so the actions are wrapped twice. `.and_then(|window| window.inner)` unwraps the outer `Option` and returns the inner one; `.unwrap_or_default()` turns `None` into an empty `Vec`.

This split also solves a borrowing puzzle. While the rack is drawing, it holds a shared borrow of the project, so it could not also change it (that would need a `&mut` at the

same time, which the borrow checker forbids). Collecting actions first and applying them after the drawing is finished keeps the two borrows apart in time.

`apply` is one `match` with an arm per action. Each arm that changes the song goes through `edit` or `edit_pattern`, which keeps the undo history and tells the audio thread what changed (see [Telling the audio thread what changed](#)):

```
RackAction::ToggleStep { channel, step } => {  
    self.edit_pattern("change steps", |pattern| {  
        pattern.toggle(channel, step);  
    });  
}
```

The same action does the same thing wherever it comes from: a click, a key, or a menu. The piano roll returns `RollAction`s and the playlist `PlaylistAction`s, and the app applies them the same way.

The selected pattern

The rack and the piano roll show **one pattern**, the selected one. The app keeps it:

```
pub struct LineupApp {  
    // ...  
    /// The selected pattern: the one the channel rack and the piano roll  
    /// show, pattern mode plays, and new playlist clips use.  
    pattern: usize,  
    mode: PlayMode,  
    // ...  
}
```

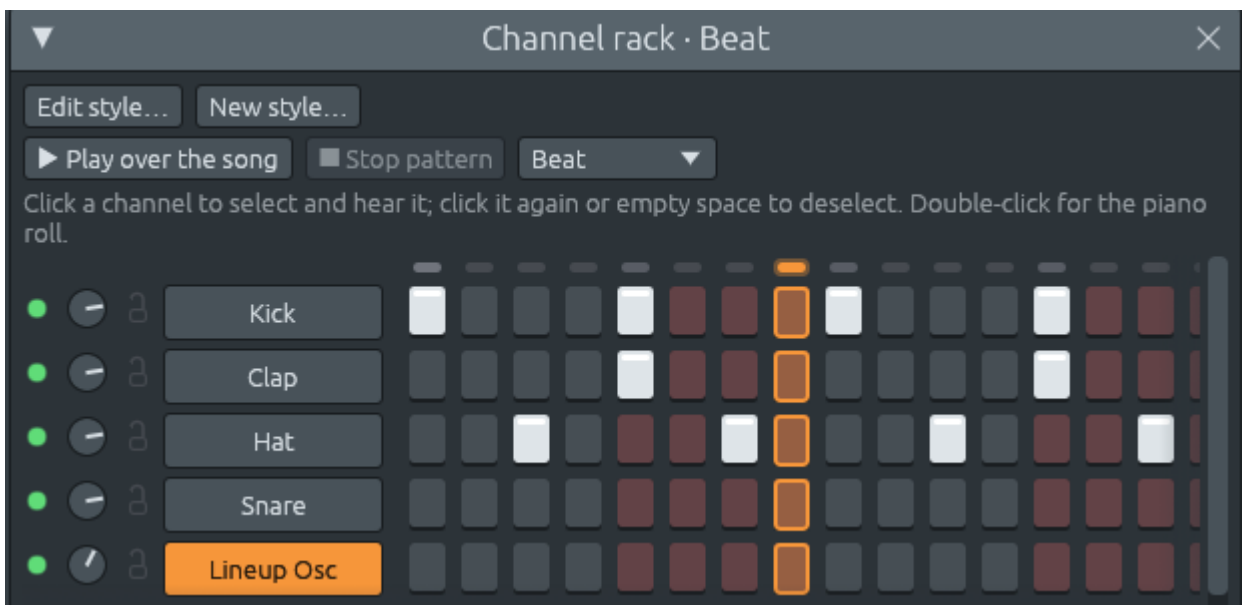
It is a field of the app, not of the project: it is *where you are looking*, not part of the song, so it is not saved and selecting a pattern does not count as an unsaved change. The editors take it as an argument, so they stay functions of what they are given.

The playhead they get depends on the mode. In pattern mode the audio thread's step is a step of the selected pattern. In song mode it is a step of the song, and the app asks the project where the selected pattern is at that moment:

```
fn pattern_playhead(&self) -> Option<usize> {
    let step = self.playhead()?;
    match self.mode {
        PlayMode::Pattern => Some(step),
        PlayMode::Song => {
            // ...a pattern played over the song gives its own step...
            let (step, _) = self.song_position()?;
            self.project.pattern_step_in_song(self.pattern, step)
        }
    }
}
```

`pattern_step_in_song` looks for a clip of the pattern, on a track that is not muted, that covers the song's step, and returns `None` if there is none. So while the song plays, the rack's playhead runs whenever a clip of the selected pattern is playing, and disappears in the bars where it is not.

The channel rack



Writing your own widgets

egui has buttons and sliders, but not step buttons or knobs. Making one takes three moves: reserve space, read the mouse, paint.

```

pub fn step_button(ui: &mut Ui, on: bool, beat_group: usize, is_playhead:
bool) -> Response {
    let (rect, response) = ui.allocate_exact_size(step_size(ui),
Sense::click());

    if ui.is_rect_visible(rect) {
        let mut fill = if on {
            STEP_ON
        } else {
            STEP_OFF[beat_group % 2]
        };
        if response.hovered() {
            fill = fill.gamma_multiply(1.25);
        }
        if is_playhead {
            // The whole column that is sounding lights up, down every
row.
            fill = fill.lerp_to_gamma(PPLAYHEAD, if on { 0.25 } else { 0.35
});
        }
        let painter = ui.painter();
        // ...a dark edge along the bottom, the pad, a light strip on a
// step that is on, an orange outline on the step playing...
    }
    response
}

```

- `allocate_exact_size` claims a rectangle in the layout and says which interactions we want (`Sense::click()`). It returns the rectangle and a `Response` describing what the mouse did there this frame. `step_size(ui)` is the step's size times the rack's zoom.
- `is_rect_visible` skips painting rows that are scrolled out of view.
- The `Painter` draws shapes. Colours are `Color32` constants defined at the top of the file, so the palette lives in one place.
- `lerp_to_gamma` mixes some orange into the step playing, a little less on a step that is on, so a lit step stays white.
- Returning the `Response` lets the caller use it like any built-in widget: `if response.clicked() { ... }`.

Beats alternate colours, so you can count them, and start again in every bar so all bars look alike: in 3/4, dark, light, dark. The rack asks the time signature which beat each step is in (`time.beat_at(step).index`), so 6/8 groups its steps in threes.

The knob uses `Sense::click_and_drag()`, turns vertical mouse movement into a value change, and calls `response.mark_changed()` so callers can use `.changed()` as they would on a slider. Its pointer is drawn with a little trigonometry: the angle sweeps

270 degrees, and `(sin, -cos)` of the angle gives the direction (screen y grows downward, hence the minus).

Menus

```
name.context_menu(|ui| self.channel_menu(ui, index, channel, locked,
actions));
```

`context_menu` opens a menu on a right-click; `menu_button` opens one on a normal click (the “Add channel” button). Both take a closure that draws the menu’s contents, and `ui.close()` closes the menu after a choice. `add_enabled_ui(false, ...)` greys out everything inside, which is how “Add channel” is disabled when the rack is full:

```
let rack_full = project.channels().len() >= MAX_CHANNELS;
ui.add_enabled_ui(!rack_full, |ui| {
    ui.menu_button(tr("✚ Add channel"), |ui| { /* ... */ });
});
```

Drag and drop

egui reports files being dragged over the window in `input.raw.hovered_files`, and files dropped in `input.raw.dropped_files`. While drawing, the rack keeps each row’s rectangle; afterwards it finds the row under the mouse:

```
let target_row = pointer.and_then(|p| row_rects.iter().position(|rect|
rect.contains(p)));
```

`position` returns the index of the first item for which the closure is true, or `None`. A WAV file dropped on a row replaces that channel’s sound; a file dropped anywhere else becomes a new channel named after the file (`path.file_stem()` is the file name without its extension). A project, MIDI or chord sheet file, or a recording, is left for the app, which opens or imports it.

Loading happens in `LineupApp::load_sample`, on the window’s thread, because reading files is slow and must never happen on the audio thread. It reads the file *before* changing anything, so a broken file leaves the channel as it was and puts the reason in the status bar.

A row of lights over the steps

A drum machine has a row of small lights, one over each step, and the light over the step being played runs along with the music. The rack has that row too. The lights must sit exactly over the steps, but the steps start after the mute light, the knob, the lock and the channel name, and their widths change with the rack's zoom. Rather than adding those widths up (and getting it wrong the next time a control is added to a row), the rack lets the rows tell it:

```
let (lights, _) = ui.allocate_exact_size(
    egui::vec2(1.0, widgets::step_lights_height(ui)),
    egui::Sense::hover(),
);
let mut steps_left = None;
for (index, channel) in project.channels().iter().enumerate() {
    let row = ui.horizontal(|ui| self.row(ui, index, channel, actions));
    steps_left.get_or_insert(row.inner);
    row_rects.push(row.response.rect);
}
if let Some(left) = steps_left {
    let marks = step_marks(self.time, self.pattern.length());
    widgets::step_lights(ui, lights, left, &marks, self.playhead);
}
```

`allocate_exact_size` reserves an empty strip at the top, so the rows go under it. Each `row` returns where its steps start (`ui.cursor().left()` just before the first step); `get_or_insert` keeps the first row's value and ignores the others. Once the rows are done, `step_lights` paints into the reserved strip. egui draws everything at the end of the frame, so painting into space reserved earlier is fine.

`step_lights` takes a slice of `StepMark`, a small enum that says whether each step starts a bar, starts a beat, or neither:

```
pub enum StepMark {
    Bar,
    Beat,
    Plain,
}
```

The light over the first step of each bar is a little brighter than the others, and the light over the first step of each beat a little brighter than the steps in between, so the row can be counted like the steps under it. Each light is placed one step width (the button plus the gap after it) after the one before, the same arithmetic the mini piano roll uses, so lights, buttons and previews line up at any zoom.

A mini piano roll for melodies

A channel whose notes are all plain steps shows step buttons. Any other channel shows `widgets::mini_roll`, a tiny drawing of its notes, and clicking it opens the piano roll. The widget has to line up with the step buttons, which have a gap between them:

```
let gap = ui.spacing().item_spacing.x;
let step_size = step_size(ui);
let step_width = step_size.x + gap;
let size = vec2(step_width * length as f32 - gap, step_size.y);
```

The pitch range is squeezed into the row's height, but never less than an octave, so three close notes do not fill the row from top to bottom:

```
let visible = || notes.iter().filter(|note| note.start < length);
let lowest = visible().map(|note| note.pitch.midi()).min().unwrap_or(0);
let highest = visible().map(|note| note.pitch.midi()).max().unwrap_or(0);
let span = (highest - lowest).max(12) as f32;
```

`visible` is a closure that makes a fresh iterator each time it is called, because `min` and `max` each use one up. The column that is playing is filled with a see-through orange (`gamma_multiply(0.25)` makes the colour a quarter as strong), as the step buttons are tinted.

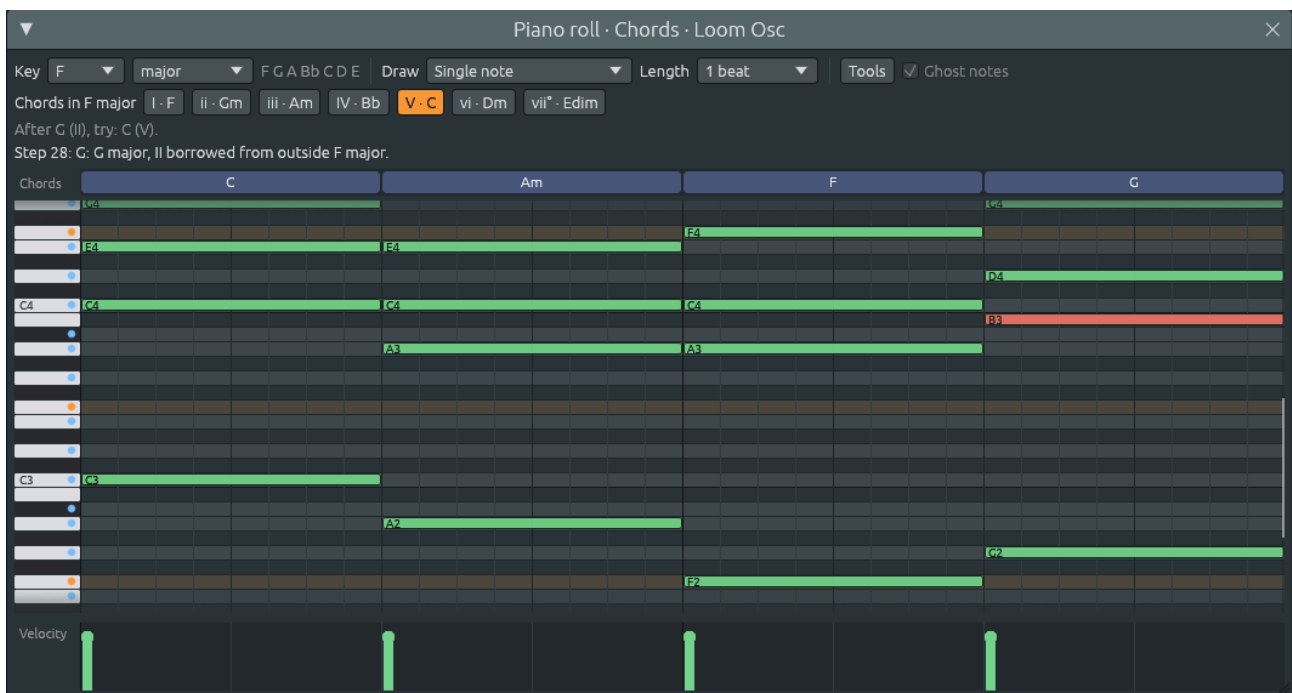
Drums in another pattern, and a rack that fits

Songs often keep their drums in a pattern of their own, “A drums” next to “A”. The rack for “A” would show drum rows with nothing in them, which looks broken. So when a pattern has no drum steps and others do, `drums_elsewhere` says so above the rows, with a button for each of those patterns.

The rack opens as big as its rows. `natural_size` adds them up (the controls, the steps, the lights, the “Add channel” menu, room for a scroll bar), and the window is given that size for one frame whenever the rows change, as [windows the size of what they show](#) do. It can still be made smaller, and then it scrolls. When every row fits, the plain mouse wheel scrolls the steps sideways. egui has a style setting for exactly that, which works when the scroll area allows only one direction:

```
ui.style_mut().always_scroll_the_only_direction = true;
let vertical = rows_height > ui.available_height();
let scroll = egui::ScrollArea::new([true, vertical])
```

The piano roll



A window with state of its own

The channel rack is a function: everything it shows comes from the project. The piano roll needs to remember things the project does not know about, from one frame to the next: which channel it shows, what a click draws, the selected notes, a drag in progress. So it is a struct with methods:

```
pub struct PianoRoll {
    pub open: bool,
    channel: usize,
    stamp: ChordStamp,
    /// The length of new notes, in steps. Resizing a note changes it, so
    /// the next note drawn is as long as the last one edited.
    note_length: usize,
    drag: Option<Drag>,
    group_drag: Option<GroupDrag>,
    selection: Vec<NoteEvent>,
    ghosts: bool,
    zoom: Vec2,
    // ...
}
```

It still never changes the project itself. `show` returns a list of `RollAction`s, as the rack does:

```

pub enum RollAction {
    /// Add one note, or the notes of a chord.
    AddNotes { channel: usize, notes: Vec<NoteEvent> },
    RemoveNote { channel: usize, index: usize },
    /// Replace a note that was moved or resized.
    ReplaceNote { channel: usize, index: usize, note: NoteEvent },
    /// Replace some notes with others, all at once: a moved selection,
    /// a tool, new velocities. `label` names the undo step.
    ReplaceNotes { channel: usize, old: Vec<NoteEvent>, new:
Vec<NoteEvent>, label: &'static str },
    /// Play some notes right now, without adding them.
    Preview { channel: usize, pitches: Vec<Note> },
    SetScale(Scale),
    /// Show another pattern: one where the channel has notes.
    SelectPattern(usize),
    /// Open the channel rack, where drum steps are edited.
    OpenRack,
    // ...
}

```

So the struct holds *view* state, which is never saved, and the project holds *document* state (the notes), which is.

The window is built like this:

```

// `open` is a copy: the window needs `&mut open` while the closure
// below needs `&mut self`, and both cannot borrow `self` at once.
let mut open = self.open;
let window = egui::Window::new(roll_title)
    .id(id)
    .open(&mut open)
    .default_pos([150.0, 160.0])
    .constrain_to(area);
// ...the size it opens at...
window.show(ctx, |ui| {
    self.contents(ui, project, pattern_notes, playhead, held, &mut
actions)
});
self.open = open;

```

- The title changes with the pattern and the channel, and egui normally identifies a window by its title. `.id(...)` gives it a fixed identity instead, so it keeps its position and size when you switch channels.
- `.open(&mut open)` adds the close button: egui sets `open` to `false` when it is clicked.
- If the window borrowed `&mut self.open` while the closure borrowed `&mut self` (to call `self.contents`), `self.open` would be borrowed mutably twice at once, which Rust forbids. Copying the `bool` out and back in avoids the overlap.

Pixels and music: Grid

Most of the drawing code turns steps and pitches into screen positions, or the pointer position back into a step and a pitch. That arithmetic lives in one place:

```
#[derive(Debug, Clone, Copy)]
struct Grid {
    /// Where all the steps and rows are, even those scrolled out of view.
    rect: Rect,
    step_width: f32,
    /// How tall a row (one pitch) is: the vertical zoom.
    row_height: f32,
    length: usize,
    time: TimeSignature,
    /// The left edge of the keyboard. The keyboard stays at the left of
    /// the view while the grid scrolls sideways under it.
    keys_left: f32,
}

impl Grid {
    /// The x coordinate where `step` begins.
    fn x(self, step: usize) -> f32 {
        self.rect.left() + step as f32 * self.step_width
    }

    /// The y coordinate of the top of a pitch's row (high notes at the
    top).
    fn row_top(self, midi: u8) -> f32 {
        self.rect.top() + (HIGHEST - midi) as f32 * self.row_height
    }

    /// The step and pitch under a point. Points outside the grid give the
    /// nearest cell, so a drag can go past the edges.
    fn cell_at(self, pos: Pos2) -> (usize, Note) {
        let row = ((pos.y - self.rect.top()) / self.row_height)
            .floor()
            .clamp(0.0, (HIGHEST - LOWEST) as f32) as u8;
        (self.step_at(pos.x), Note::new(HIGHEST - row))
    }

    // ...keys, cells, step_at, note_rect, text_size
}
```

High notes are at the top, so the row counts down from `HIGHEST`. `cell_at` clamps, so a drag that leaves the grid still gets the nearest cell, and returns a **tuple**, `(usize, Note)`, which the caller takes apart with `let (step, pitch) = grid.cell_at(pos);`. `note_rect` draws a note from its exact start to its exact end, including how far it is nudged off the step, so a triplet sits between two steps, as it is heard.

Because `Grid` is `Copy` and its methods take `self` by value, it can be passed to every drawing function without any `&` or lifetimes. For a struct of a few numbers, copying is

as cheap as passing a reference.

At 100% the whole pattern fits across the window, so the step width is worked out on every frame from the window's width. Zooming in makes the steps wider, and the grid scrolls sideways under the keyboard.

A keyboard that stays, lanes that follow

The grid is inside a scroll area, but two things must not scroll away with it. The piano keys are drawn at the left edge of what is *visible* (`keys_left: visible.left()`), on top of the grid, so the notes slide under them. The chord lane above the grid and the velocity lane below it are drawn outside the scroll area, so they do not know how far the grid is scrolled. A small struct carries what they need to line up:

```
/// What the chord and velocity lanes need to line up with the grid.
#[derive(Debug, Clone, Copy)]
struct Lanes {
    step_width: f32,
    /// How far the grid is scrolled sideways, in pixels.
    scroll_x: f32,
    length: usize,
}
```

Each lane builds its own `Grid` from it (`Lanes::grid`), moved by `scroll_x`, and uses the same functions as the note grid to find where a step is. One source of truth for “where is step 12?” is what keeps three strips drawn separately in line.

Mouse input as a small state machine

egui reports what the mouse is doing *this frame*. A drag lasts many frames, so the piano roll remembers it in an enum:

```
enum Drag {
    /// Moving a note. `grab` is how many steps after the note's start the
    /// pointer took hold of it, so the note does not jump under the
    pointer.
    Move { index: usize, grab: usize },
    /// Changing a note's length by dragging its right end.
    Resize { index: usize },
}
```

`handle_input` is long, but each part answers one question:

1. **Was the left button just pressed on the grid?** If it hit a note, start a `Move`, or a `Resize` if the press was near the note's right end. If it hit an empty cell, add what a click draws (a note or a chord); a single new note also starts a `Move`, so you can place and drag in one gesture.
2. **Is the right button down over a note?** Delete it. Checking every frame means you can sweep across several notes to delete them all.
3. **Is the left button up?** Then any drag is over.
4. **Is a drag in progress?** Work out where the note would be now, and if that differs from where it is, send a `ReplaceNote`.

The fourth part uses a **let chain** with three conditions, and struct update syntax to change one or two fields:

```
if let Some(drag) = self.drag
  && let Some(pos) = pointer
  && let Some(&note) = notes.get(drag.index())
{
  let (step, pitch) = grid.cell_at(pos);
  let pitch = self.snapped(pitch, scale);
  let changed = match drag {
    Drag::Move { grab, .. } => NoteEvent {
      pitch,
      start: step.saturating_sub(grab),
      ..note
    },
    Drag::Resize { .. } => {
      let length = (step + 1).saturating_sub(note.start).max(1);
      self.note_length = length;
      // ...
      NoteEvent { length, trim, ..note }
    }
  };
  if changed != note {
    // ...play the new pitch, keep the note selected...
    actions.push(RollAction::ReplaceNote { channel, index:
drag.index(), note: changed });
  }
}
```

`Some(&note)` in a pattern copies the note out of the `&NoteEvent` that `get` returns. Comparing `changed != note` (thanks to the derived `PartialEq`) means nothing is sent while the pointer stays inside the same cell, even though this code runs every frame. `snapped` moves the pitch onto the key when **Snap to scale** is on.

Two details make clicks land where they were meant. A quick right-click can press and release the button within a single frame, so “is the button down?” is already false by the time the frame is drawn. The code asks “was it pressed this frame, or is it down?” instead:

```
pointer.secondary_pressed() || pointer.secondary_down(),
```

And on the frame the left button goes down, the press is handled where the button *went down*, which `widgets::press_pos` reads from the frame's events (see [Clicks that do what you meant](#)): a quick hand can press and move on in the same frame.

When the pointer is over an empty cell, the piano roll draws outlines of what a click would place, from the same `stamp_notes` the click itself uses. With “Chord in key (seventh)” chosen, you see the whole chord before you place it.

The selection holds notes, not positions

The simplest way to remember which notes are selected would be their positions in the channel's list: `selection: Vec<usize>`. The single-note drag does exactly that (`Drag::Move { index, .. }`), and it works there because nothing else changes the list during a drag.

A selection lives much longer than a drag. While notes are selected, a tool takes them out of the list and puts new ones at the end, undo swaps in a whole older project, and deleting a note moves every note after it one place up. After any of those, position 3 may be a different note, or no note at all. A selection of positions would quietly select the wrong notes, and the next tool would change notes you never chose.

So the selection holds the notes themselves:

```
/// The selected notes, by value. A note that is no longer in the
/// pattern (after an undo, say) is simply not found, and so not
/// selected: the selection can never point at the wrong note.
selection: Vec<NoteEvent>,
```

This works because a channel never holds two notes in the same place: the pattern's rule (same pitch, same start, same nudge) makes every note's value unique, so a value names exactly one note. `NoteEvent` is `Copy` and `PartialEq`, so keeping copies and comparing them is cheap and easy.

At the start of every frame, notes that have gone are forgotten:

```
let notes = pattern.notes(self.channel);
self.selection.retain(|note| notes.contains(note));
```

And when the piano roll replaces some notes, it makes the selection follow them, so that pressing `↑` three times moves the same chord three times:

```

fn replace(
    &mut self,
    pattern: &Pattern,
    actions: &mut Vec<RollAction>,
    label: &'static str,
    old: Vec<NoteEvent>,
    new: Vec<NoteEvent>,
) -> Option<Vec<NoteEvent>> {
    let new = tidy_notes(new);
    if old == new || !pattern.can_replace_notes(self.channel, &old, &new)
    {
        return None;
    }
    let was_selected = old.iter().any(|note|
self.selection.contains(note));
    self.selection.retain(|note| !old.contains(note));
    if was_selected {
        self.selection.extend(&new);
    }
    actions.push(RollAction::ReplaceNotes {
        channel: self.channel,
        old,
        new: new.clone(),
        label,
    });
    Some(new)
}

```

`replace` is the one door every group change goes through: the tools, the group drag and the velocity lane. It tidies the new notes the same way the pattern will, asks the pattern whether the change is allowed, and only then sends the action. Tidying here too means the selection holds exactly the notes that will be in the pattern, and checking first means a refused move does nothing at all instead of half a thing.

The comparison costs a little: `contains` looks through the whole list, so checking every note against the selection takes time proportional to the number of notes times the size of the selection. For a piano roll with a few hundred notes, that is far below a millisecond. If it ever mattered, a `HashSet<NoteEvent>` would make each check constant time (`NoteEvent` already derives `Hash` for that).

Selecting with the mouse

Ctrl+click on a note adds it to the selection, or takes it out:


```

if let Some(at) = self.selection.iter().position(|&n| n == note) {
    self.selection.remove(at);
} else {
    self.selection.push(note);
}

```

`input.modifiers.command` is “Ctrl” on Linux and Windows and “⌘” on a Mac, so the same code follows each system’s habits.

Ctrl+drag on empty space draws a box. Only the corner where it started is kept; the selection is rebuilt from scratch on every frame, from that corner and the pointer, so making the box smaller again unselects notes:

```

if let Some(origin) = self.selection_box
    && let Some(pos) = pointer
{
    let area = Rect::from_two_pos(origin, pos);
    self.selection = notes
        .iter()
        .filter(|note| note.start < grid.length &&
grid.note_rect(note).intersects(area))
        .copied()
        .collect();
}

```

`Rect::from_two_pos` makes a rectangle from any two corners, whichever way the box was dragged, and `intersects` is true when two rectangles overlap at all, so a note only needs to be touched by the box.

Moving a group

Pressing a selected note (anywhere but its right end, which still resizes) starts a group drag:

```

struct GroupDrag {
    /// The cell where the drag started.
    from_step: usize,
    from_pitch: Note,
    /// The notes as they were when the drag started.
    originals: Vec<NoteEvent>,
    /// The notes as they are now, after the moves so far.
    moved: Vec<NoteEvent>,
    /// Whether the notes have moved at all.
    edited: bool,
}

```

On every frame of the drag, the new position is worked out from the *originals*, with the whole distance from the start cell to the pointer:

```
let steps = step as isize - drag.from_step as isize;
let semitones = i32::from(pitch.midi()) -
i32::from(drag.from_pitch.midi());
drag.edited |= steps != 0 || semitones != 0;
let snap = self.snap_to_scale && drag.edited;
let new = tools::shift(&drag.originals, steps, grid.length)
    .and_then(|shifted| tools::transpose(&shifted, semitones))
    .map(|moved| match snap {
        true => tools::snap_to_scale(&moved, scale),
        false => moved,
    })
    .filter(|moved| fits_view(moved));
```

Moving the notes a little each frame (from `moved`) would be the obvious way, but small errors would add up: a move refused at the edge of the pattern would be lost, and the notes would end up somewhere other than under the pointer. Starting from the originals every time means the notes are always exactly where the pointer says, and dragging back to the start puts them back exactly as they were.

The tools (see [Editing tools as plain functions](#)) are joined with `Option::and_then`: shift, and *if that worked*, transpose the result. `map` snaps the notes to the key when Snap to scale is on, but only once they have moved, so dragging a selection up a semitone and back down snaps it into the key. `filter` then refuses notes outside the rows the piano roll shows. The replacement goes from `moved` (what is in the pattern now) to `new`, and `moved` is updated to what `replace` returned.

The drag is taken out of `self` while it is worked on:

```
if let Some(mut drag) = self.group_drag.take()
    && let Some(pos) = pointer
{
    // ... self.replace(pattern, actions, "move notes",
    drag.moved.clone(), new) ...
    self.group_drag = Some(drag);
}
```

`self.replace` needs `&mut self`, the whole struct. If `drag` were a reference into `self.group_drag`, the borrow checker would refuse: one part of `self` would be borrowed while all of `self` is borrowed mutably. `Option::take` moves the drag out and leaves `None`, so `drag` is a separate value and `self` is free; the drag is put back at the end.

Every frame of the drag produces a `ReplaceNotes`, and each one goes through `edit_pattern`. They still make a single undo step, because a step stays open while a mouse button is held (see [Undo and redo](#)). And because the step is already open, `edit` does not copy the whole project again on every frame; it sends the audio thread what changed at most every 50 ms, and the rest when the button is released. Nothing in the piano roll has to know about either.

Who gets the keys

The arrow keys and Delete are useful in the piano roll, but the rack and the playlist use them too, and a key that changes notes in a window you are not looking at would be a nasty surprise. So the editor keys go to the window clicked last, and to no other:

```
/// Editor shortcuts belong to the last clicked window, including its
title
/// and toolbar. Popup controls and text fields keep their own keyboard
input.
pub fn editor_keyboard(ctx: &egui::Context, layer: egui::LayerId) -> bool
{
    editor_active(ctx, layer)
        && !ctx.text_edit_focused()
        && !egui::Popup::is_any_open(ctx)
        && ctx.memory(|memory| memory.is_above_modal_layer(layer))
}
```

`editor_active` remembers the *layer* of the last press (each window is a layer, see [What is under the pointer?](#)), in egui's memory, and answers whether it is this one. A window the app opens for you, as a double-click on a clip opens the piano roll, is given the keyboard with `widgets::give_keyboard`, as if it had been clicked. The playlist shares the background layer with the toolbars, so for it the place of the click matters too (`editor_keyboard_in`). The keys are ignored while a text field has the keyboard, so typing a pattern name never moves notes, and while a menu is open.

Space, which plays and stops, has the opposite problem: it must work everywhere, but egui lets a focused button be pressed with Space too. Click ► Play, then press Space, and the shortcut would stop the music while the button started it again. So the shortcuts are read before any panel is drawn, and `consume_key` takes the press away once it is used:

```

let typing = ctx.text_edit_focused() || egui::Popup::is_any_open(ctx);
// `consume_key` reports whether the key was pressed and removes the
// press, so a focused button does not also react to Space.
let pressed = |key| ctx.input_mut(|input|
input.consume_key(egui::Modifiers::NONE, key));
if !typing && !controls_active && pressed(egui::Key::Space) {
    // ...play or stop the song, or the pattern in the rack or piano
    roll...
}

```

The order of the shortcuts

Each key is a `KeyboardShortcut`: a key and its modifiers.

```

pub(crate) const KEY_UP: KeyboardShortcut = shortcut(Modifiers::NONE,
Key::ArrowUp);
pub(crate) const OCTAVE_UP: KeyboardShortcut = shortcut(Modifiers::SHIFT,
Key::ArrowUp);
pub(crate) const IN_KEY_UP: KeyboardShortcut = shortcut(Modifiers::ALT,
Key::ArrowUp);

```

`shortcut` is a `const fn`, a function simple enough to run while compiling, which is why its results can be `const` s. They are `pub(crate)` so that the list in **Help > Keyboard shortcuts** uses the same constants (see [Every key in a list](#)).

The piano roll asks egui whether each one was pressed with

`input.consume_shortcut(&shortcut)`. That returns `true` once, and “consumes” the key press so nothing else reacts to it. One detail of egui matters here: it matches shortcuts *logically*, which means extra Shift or Alt keys do not stop a match. Shift+↑ counts as a press of ↑ as well. Checked in the wrong order, Shift+↑ would be eaten by the “up a semitone” shortcut before the “up an octave” one ever saw it. So the list goes from the most specific to the least:

```

// egui counts Shift+↑ and Alt+↑ as ↑ too, so the longer shortcuts
// are checked first.
let keys = [
    (IN_KEY_UP, Tool::InKey(1)),
    (IN_KEY_DOWN, Tool::InKey(-1)),
    (OCTAVE_UP, Tool::Transpose(12)),
    (OCTAVE_DOWN, Tool::Transpose(-12)),
    (KEY_UP, Tool::Transpose(1)),
    // ...
];
keys.into_iter()
    .find(|&(shortcut, _)| pressed(shortcut))
    .map(|(_, tool)| tool)

```

`find` stops at the first shortcut pressed, so at most one tool runs per frame, and the later, shorter shortcuts are not even asked.

One enum for the menu and the keys

The Tools menu and the keys both produce a `Tool`:

```
enum Tool {  
    Copy,  
    Cut,  
    Paste,  
    Duplicate,  
    Transpose(i32),  
    InKey(i32),  
    Shift(isize),  
    Quantize,  
    SnapToKey,  
    Invert { up: bool },  
    Arpeggiate,  
    Delete,  
    SelectAll,  
    SelectNone,  
}
```

and `use_tool` turns any `Tool` into a call to the right function in `lineup_core::tools`, followed by `replace`. A tool changes the selected notes, or every note when none is selected; Delete only ever deletes a selection. Whether a tool came from the menu or a key, it takes exactly the same path, so the two can never behave differently. The menu shows each tool's key next to it, written the way the system writes keys, so it doubles as the list of shortcuts.

A menu that stays open

egui menus close on any click inside them, which is right for a list of commands. But the Tools menu also holds choices: the arpeggio's order and speed. Closing the menu when you pick "up and down" would make you open it again to press "Arpeggiate chords". So this menu is told to close only on a click *outside* it, and each tool closes it by hand:

```

let stay_open = egui::containers::menu::MenuConfig::new()
    .close_behavior(egui::PopupCloseBehavior::CloseOnClickOutside);
let menu =
egui::containers::menu::MenuButton::new(tr("Tools")).config(stay_open);
menu.ui(ui, |ui| {
    // ...
    let mut item =
        |ui: &mut Ui, text: &str, tool: Tool, shortcut:
Option<KeyboardShortcut>| {
        // ...
        if ui.add(button).clicked() {
            picked = Some(tool);
            ui.close();
        }
    };
    item(ui, "Up a semitone", Tool::Transpose(1), Some(KEY_UP));
    // ...
});

```

`item` is a closure stored in a variable and called many times. It assigns to `picked`, a variable outside it, so it borrows `picked` mutably for as long as the closure exists, and has to be declared `let mut item` (calling it changes what it captured: it is an `FnMut`). The `ui` is passed in as an argument on each call instead of being captured, because the code between the calls uses `ui` too, and a closure holding on to it would block that. Menus with a text field in them, such as the playlist's section menu, stay open the same way, so a click into the field does not close them.

Ghost notes: hearing the other channels

A chord is often shared between instruments: the pad plays C and G, the bass plays E, and together they make C/E. Looking only at the channel being edited, the piano roll would say "C5" (a power chord) where you hear C/E.

So the piano roll can listen to the other channels as well. Not all of them: drum channels hold plain steps, and a kick drum is not part of the harmony. A channel counts if any of its notes is not a plain step:

```

let others: Vec<usize> = (0..pattern.rows())
    .filter(|&other| other != self.channel)
    .filter(|&other| pattern.notes(other).iter().any(|note|
!note.is_step()))
    .collect();
let mut heard = vec![self.channel];
if self.ghosts {
    heard.extend(&others);
}

```

`heard` is the list of channels whose notes count for the harmony: this one, and the others when “Ghost notes” is ticked. The line that names the chord under the pointer, the next-chord hints and the note names all take `heard`, and gather its notes with a small helper:

```
fn sounding<'a>(  
    pattern: &'a Pattern,  
    channels: &'a [usize],  
    step: usize,  
) -> impl Iterator<Item = Note> + 'a {  
    channels  
        .iter()  
        .flat_map(move |&channel| pattern.sounding_at(channel, step))  
}
```

- `flat_map` calls the closure for each channel, gets an iterator of notes back each time, and joins them all into one iterator, as if the channels’ notes were one list.
- The result borrows `pattern` and `channels`, so it must not outlive them. `<'a>` names that borrow, and `+ 'a` on the return type says “this iterator lives no longer than `'a`”. `analyze` takes any iterator of notes, so it reads the notes straight from the pattern, without copying them into a `Vec` first.
- `move` makes the closure take its own copies of what it uses: `step` (a number) and `pattern` (a reference, which is copied, not the pattern). Without `move`, the closure would borrow the variable `step`, which belongs to `sounding` and disappears when it returns, while the iterator lives on.

The ghost notes are painted before this channel’s notes, so this channel’s notes are always on top, in grey with a thin border. They are only drawn: clicks only look at this channel’s notes, so ghost notes can never be selected, moved or deleted by accident. When no other channel plays melodies or chords, the checkbox is greyed out, and its hover text says why.

The chord lane: `chunk_by`, then `measure`

The lane above the grid names the harmony. Its first row names what this channel plays at every step; its second row, **All**, names what the whole song sounds like there, beat by beat, as [Hearing harmony](#) works it out. Each row joins neighbouring steps with the same harmony into one block:

```
// `chunk_by` splits a slice wherever two neighbours differ. A
// harmony holds its bass, so C and C/E are two blocks.
let mut start = 0;
for run in harmonies.chunk_by(|a, b| a == b) {
    let end = start + run.len();
    if let Some(label) = run[0].label_in(scale) {
        // ...a block from step `start` to step `end`, with the label...
    }
    start = end;
}
```

`chunk_by` splits a slice into runs wherever the closure says two neighbours do not belong together. Each `run` is a slice of the original vector, so nothing is copied. Comparing whole harmonies, not their labels, matters: every group of notes without a chord name is labelled “?”, and joining by label would turn several different unnamed chords in a row into one long “?” block, as if one chord were held.

Running `analyze` for every step on every frame sounds like a lot, but it is at most a few hundred small loops over a few notes. Measure before optimizing: here it does not show up at all.

A name must fit its block. Cut off at the edges, “E7/B” in a narrow block would show “7/”, which is not a name at all, and for someone learning harmony from the screen a wrong name is worse than none. `egui` draws text in two steps. First it **lays it out**: it picks the glyph of each character in the font, and works out where each goes and how much room the whole takes. The result is a *galley* (the name comes from printing: a tray of type set in lines, ready to print). Then it paints the galley. `painter.text(...)` does both at once; to decide *before* drawing, the two steps are taken apart:

```
let name = painter.layout_no_wrap(label, FontId::proportional(11.0),
LANE_TEXT);
if name.size().x <= chord_rect.width() - 2.0 {
    // A long chord's name stays in view while it scrolls.
    let shown = chord_rect.intersect(cells);
    painter.with_clip_rect(chord_rect).galley(
        shown.center() - name.size() / 2.0,
        name,
        LANE_TEXT,
    );
}
```

`layout_no_wrap` lays the text out on one line, and `name.size()` is how wide and tall it is. If it fits in the block, with a point to spare on each side, the same galley is painted; nothing is laid out twice. `galley` takes the top-left corner of the text, so the centre is moved back by half the size. If it does not fit, nothing is drawn; pointing at the block still names the chord in full in the line under the toolbar.

Scale highlighting

`paint_rows` colours each row by asking the scale about its pitch class, with a `match` on a tuple of two `bool`s:

```
let fill = match (is_root, in_scale) {  
    (true, _) => ROOT_ROW,  
    (false, true) => SCALE_ROW,  
    (false, false) => OTHER_ROW,  
};
```

The keys on the left get a dot for notes in the key (orange for the root), and notes outside the key are drawn red. The key is the song's key where the pattern is played (`project.pattern_key`), so a pattern used after a key change is shown in the new key. Change the key and every colour follows on the next frame, because nothing is cached: immediate mode draws from the current state every time.

The velocity lane

Making room for it

The lane sits under the grid. The grid is a `ScrollArea` that fills the height it is given. Given all the height left in the window, it would push the lane below the window's edge; egui windows grow to fit their contents, so the window would grow, the grid would fill the new height, the lane would be pushed out again, and the window would creep a few pixels taller on every frame. So the grid is given exactly the height that is left once the lane has its share:

```
// The velocity lane goes below the grid, so the grid leaves room  
// for it: the lane, the gap, and egui's spacing between widgets.  
// Taking even a pixel too much would make the window grow a  
// little on every frame.  
let below = VELOCITY_HEIGHT + 4.0 + 2.0 * ui.spacing().item_spacing.y;  
let grid_height = (ui.available_height() - below).max(96.0);
```

The easy part to forget is `item_spacing`: egui puts a small gap after every widget, here after the scroll area and after the space. A simple way to catch a window that creeps is to take two screenshots a few seconds apart and compare them.

Setting velocities

The lane is one widget that senses clicks and drags. While the button is down on it, the pointer's height sets the velocity, and its x position picks the step:

```
if response.is_pointer_button_down_on()
    && !crate::widgets::dismissing(ui.ctx())
    && let Some(pos) = response.interact_pointer_pos()
    && (cells.contains(pos) || response.dragged())
{
    let step = lane.step_at(pos.x);
    let height = (lane.rect.bottom() - pos.y) / lane.rect.height();
    let velocity = (height * 127.0).round().clamp(1.0, 127.0) as u8;
    let selected_here = notes
        .iter()
        .any(|note| note.start == step && self.selection.contains(note));
    let old: Vec<NoteEvent> = notes
        .iter()
        .filter(|note| note.start == step && note.velocity != velocity)
        .filter(|note| !selected_here || self.selection.contains(note))
        .copied()
        .collect();
    let new = old.iter().map(|&note| NoteEvent { velocity, ..note
}).collect();
    self.replace(pattern, actions, "change velocity", old, new);
}
```

- The height is a fraction from 0 (bottom) to 1 (top), scaled to MIDI's 1 to 127. `clamp` keeps it inside even when the pointer leaves the lane during a drag. Casting a float with `as` never wraps (300.0 becomes 255 and -5.0 becomes 0), but 0 would be a note you cannot hear, so the clamp makes sure the velocity is at least 1.
- A chord has several notes on the same step. All of them change, unless some of them are selected: then only the selected ones do. That is how you make the top note of a chord louder than the others, the way pianists bring out a melody.
- Notes that already have the velocity are left out of `old`, so holding the button still does not send the same change every frame.

Sweeping across the lane changes each step as the pointer passes over it, and the whole sweep is one undo step, like any drag.

Velocity shows in the grid too. A note's colour is multiplied by a factor between 0.55 and 1 depending on its velocity (`fill.gamma_multiply(0.55 + 0.45 * f32::from(note.velocity) / 127.0)`), so soft notes are darker, but never so dark that they disappear.

Finding your notes

The piano roll shows one channel of one pattern, and that pair can be empty while the song plays plenty: the chords of a song may be in “Chords”, not in “Beat”. An empty grid with no word of explanation looks like lost notes. So the piano roll does two things.

First, it follows the notes. When it opens, or the pattern changes, and the channel shown has no notes there, it moves to the channel with the most melody or chord notes:

```
pub fn busiest_channel(pattern: &Pattern) -> Option<usize> {
    (0..pattern.rows())
        .map(|channel| {
            let melodic = pattern.notes(channel).iter().filter(|note|
!note.is_step());
            (channel, melodic.count())
        })
        .filter(|&(_, count)| count > 0)
        .max_by_key(|&(channel, count)| (count,
std::cmp::Reverse(channel)))
        .map(|(channel, _)| channel)
}
```

`max_by_key` returns the *last* of several equal maximums. Comparing the tuple `(count, Reverse(channel))` breaks ties towards the lower channel number instead: `std::cmp::Reverse` wraps a value and flips its ordering. Drum steps are left out on purpose: in a pattern of drums alone, the piano roll does not jump to the kick drum, whose “notes” are all the same plain step. A channel you picked yourself, in the rack or the editor, is kept.

To notice the window *opening*, the piano roll remembers whether it was open last frame. `open` is a public field that the toolbar button, F7 and the View menu flip, so this is the one place that sees every way in:

```
let opened = self.open && !self.was_open;
// ... at the end of `show`:
self.was_open = open;
```

On opening, it also scrolls to the middle of the notes. The middle is not worked out by halving pitches: with integers, `/` throws away the remainder, and `41 / 2 + 67 / 2` is 53, a row off the true middle, 54. Instead it asks egui to centre the whole block of rows, from the top of the highest note to the bottom of the lowest:

```
let top = grid.row_top(highest.clamp(LOWEST, HIGHEST));
let bottom = grid.row_top(lowest.clamp(LOWEST, HIGHEST)) + row_height;
let target = Rect::from_min_max(pos2(rect.left(), top), pos2(rect.left() + 1.0, bottom));
ui.scroll_to_rect(target, Some(Align::Center));
```

Second, when the channel really has no notes in the pattern, it says so over the grid, with buttons to go where the notes are:



Finding the patterns where the channel plays is a filter over all of them, keeping the index (to select it) and the name (to show it):

```
let elsewhere: Vec<(usize, &str)> = project
    .patterns()
    .iter()
    .enumerate()
    .filter(|&(index, other)| {
        index != self.pattern && !other.notes(self.channel).is_empty()
    })
    .map(|(index, other)| (index, other.name.as_str()))
    .collect();
```

The `&str` in the vector borrows from `project`. That is fine because the vector lives only while this frame is drawn, and `project` outlives it. A button for another pattern cannot select it directly: the selected pattern belongs to the app. So it returns `RollAction::SelectPattern`, the same way every other change leaves the piano roll.

The message lies *over* the grid instead of taking a line above it, so the grid does not jump up when the first note is drawn and the message goes away. It is drawn after the grid, in a child `ui` placed over it, and the grid remembers where it was drawn last frame (`self.banner`) and leaves clicks there alone: a click on one of its buttons must not also draw a note underneath.

Wiring it into the app

The app owns a `PianoRoll` and, while drawing the windows, calls:

```
let roll_actions = self.piano_roll.show(
  ctx,
  &self.project,
  self.pattern,
  playhead,
  &held,
  area,
  roll_zoom,
);
for action in roll_actions {
  self.apply_roll(action);
}
```

`self.piano_roll.show(..., &self.project, ...)` borrows one field mutably and another one immutably at the same time, which the borrow checker allows because they are different fields. `apply_roll` then changes the pattern through `edit_pattern`, as the rack does:

```
RollAction::ReplaceNotes { channel, old, new, label } => {
  self.edit_pattern(label, |pattern| {
    pattern.replace_notes(channel, &old, &new);
  });
}
```

When a channel is removed, `PianoRoll::channel_removed` either lets go of its channel (it is gone) or moves its channel number down by one, so it keeps showing the same channel; `channel_inserted` does the opposite when undo brings a channel back.

In a short window, under 420 points, the grid comes first: the hint lines under the toolbar move into the tooltip of the line that names the chord, so a small screen still shows a dozen rows of notes.

The playlist

A big grid: show_viewport

The playlist can be far bigger than the window: dozens of bars, and as many tracks as the song uses. It goes in a scroll area. But some parts must *not* scroll away: the track names on the left, and the bar numbers and the strips above them at the top.

`ScrollArea::show_viewport` makes this possible. Its closure receives the visible part of the content, as a rectangle measured from the content's top-left corner:

```
let output = area.show_viewport(ui, |ui, viewport| {
    self.grid(ui, viewport, project, current, playhead, &mut actions)
});
```

`grid` allocates the full size of the content, then works out where the visible part is on screen:

```
let (rect, response) = ui.allocate_exact_size(size,
Sense::click_and_drag());

// `viewport` is the visible part, measured from the top-left of the
// content. Moving it by the content's position on screen gives the
// visible part in screen coordinates, where everything is drawn.
let visible = viewport.translate(rect.min.to_vec2());
```

`rect` is the whole content in screen coordinates (its top-left corner moves up and left as you scroll). Adding that corner to `viewport` gives `visible`, the part of the screen where the content shows. Then:

- the tracks and clips are drawn at their place in the content, so they scroll;
- the track names are drawn at `visible.left()`, and the ruler and the strips at `visible.top()`, so they stay put while the clips slide underneath.

They are drawn *after* the clips, so they cover them.

The arithmetic for steps, tracks and pixels lives in a small `Copy` struct, as in the piano roll:

```
#[derive(Debug, Clone, Copy)]
struct Layout<'a> {
    /// The top-left corner of step 0 of track 0.
    origin: Pos2,
    /// How wide a step is and how tall a track is: the zoom.
    step_width: f32,
    track_tops: &'a [f32],
    tracks: usize,
    /// Whether there is an empty row below the last track, where a clip
    /// starts a new track. There is unless the playlist is full.
    spare: bool,
}
```

Each track can have its own height, so `Layout` keeps where each track starts (`track_tops`), worked out once per frame. To find the track under the pointer, `cell_at` uses `partition_point`, a binary search on that sorted list: the number of tops above the pointer, minus one, is the track.

Real widgets in a painted area: `ui.put`

Everything in the grid is painted by hand, but the track names should be editable text fields and the mute lights should be the same light as in the rack. `ui.put` places any widget in a rectangle you choose:

```
let light_response = ui
    .put(light, |ui: &mut Ui| widgets::mute_light(ui, enabled))
    .on_hover_text(tr("Click to mute or unmute the track"));

let shown = tr_name(&track.name).into_owned();
let mut name = shown.clone();
let field = egui::TextEdit::singleline(&mut name)
    .frame(egui::Frame::NONE)
    .text_color(HEADER_TEXT);
let name_response = ui.put(name_rect, field);
if name_response.changed() && name != shown {
    actions.push(PlaylistAction::RenameTrack { track: index, name });
}
```

- A closure `|ui: &mut Ui| -> Response` counts as a widget: `egui` implements its `Widget` trait for any such closure. That is how a function-style widget like `mute_light` can be passed to `put`.
- The text field edits a *copy* of the name. If it changed, the copy goes into a `RenameTrack` action, and the app changes the project. The playlist keeps the rule that it never changes the project itself.
- `Frame::NONE` removes the field's border, so it looks like a label until you click it.

Placing, moving and stretching clips

Placing, moving and deleting clips follows the piano roll's pattern: a press decides what the drag is, and the drag follows the pointer until the button is released.

```
struct ClipDrag {  
    index: usize,  
    grab: usize,  
    stretch: bool,  
}
```

A press on empty space places a clip of the selected pattern and starts moving it; a press on a clip starts moving that clip and selects its pattern; a press on a clip's right edge starts stretching it, which repeats its pattern as many times as fit up to the pointer. The press is handled where the button went down:

```
// A press is handled where the button went down. A quick hand can  
// press and move in the same frame, and the clip or empty cell  
// under the pointer *now* may not be the one that was clicked.  
// A click that closes a menu does nothing else.  
let dismissing = widgets::dismissing(ui.ctx());  
let pressed_at = press_origin.filter(|pos| {  
    primary_pressed && !dismissing && response.hovered() &&  
    cells.contains(*pos)  
});  
if let Some(pos) = pressed_at {  
    let (track, step) = layout.cell_at(pos);  
    match clip_at(project, layout, pos) {  
        Some(index) => { /* start moving or stretching that clip */ }  
        None => { /* add a clip of the selected pattern, and start moving  
it */ }  
    }  
}
```

`Option::filter` keeps the value only if the closure returns `true`, so `pressed_at` is `Some` only on the frame of a press, over the grid.

Positions snap to the grid you choose in the toolbar: a bar, a beat or a step. Where bars and beats fall depends on the time signature at that point of the song, so the snap asks the song's `Meter`:


```

/// The snap point at or before `step`. Where the bars and beats are
/// depends on the time signature there: a bar of 3/4 is 12 steps, a
/// beat of 6/8 is 6.
fn floor(self, step: usize, meter: &Meter) -> usize {
    match self {
        Snap::Bar => meter.floor_to_bar(step),
        Snap::Beat => meter.floor_to_beat(step),
        Snap::Step => step,
    }
}

```

Ctrl or Shift and a click adds a clip to the selection, Ctrl and a drag on empty space draws a box around clips, and dragging one of several selected clips moves them all, kept inside the playlist.

Double-click a clip to edit it

In most music programs you double-click a clip to edit it. egui counts the clicks:

```

let double_clicked = ui.input(|input| {
    input
        .pointer
        .button_double_clicked(egui::PointerButton::Primary)
});
if double_clicked
    && !dismissing
    && response.hovered()
    && let Some(pos) = pointer.filter(|pos| cells.contains(*pos))
    && let Some(index) = clip_at(project, layout, pos)
{
    actions.push(PlaylistAction::EditClip(clips[index].pattern));
}

```

The app picks the editor: the piano roll, on the busiest channel, when the pattern has melodies or chords; the channel rack when it only has drums. Either way the window comes to the front, and takes the keyboard with it:

```

fn bring_window_to_front(&mut self, ctx: &egui::Context) {
    if let Some(id) = self.bring_to_front.take() {
        let layer = egui::LayerId::new(egui::Order::Middle, id);
        ctx.move_to_top(layer);
        // Its keys go with it: Delete after double-clicking a clip
        // deletes notes in the piano roll, not clips in the playlist.
        widgets::give_keyboard(ctx, layer);
    }
}

```

A window's layer is named by the id the window was given, which is why the rack and the piano roll get theirs from small functions (`rack_id()` and `PianoRoll::id()`) instead of writing the same `Id::unique(...)` in several places.



Drawing a clip's notes: Box<dyn Fn>

Each clip shows a sketch of its pattern. Drum patterns look best as a tiny step sequencer (one line per channel), anything with melody as a tiny piano roll (higher notes higher). The two need different ways to turn a note into a height, chosen once per clip:

```
// Turns a channel and a note into a height from 0.0 (bottom) to 1.0
(top).
let height: Box<dyn Fn(usize, u8) -> f32> = if steps_only {
    let used: Vec<usize> = (0..channels)
        .filter(|&channel| !pattern.notes(channel).is_empty())
        .collect();
    let count = used.len().max(1) as f32;
    Box::new(move |channel, _| {
        let row = used.iter().position(|&c| c == channel).unwrap_or(0) as
f32;
        1.0 - (row + 0.5) / count
    })
} else {
    // ...the lowest and highest note, as in the rack's mini piano roll...
    Box::new(move |_, midi| ((midi - lowest) as f32 + centering) / span)
};
```

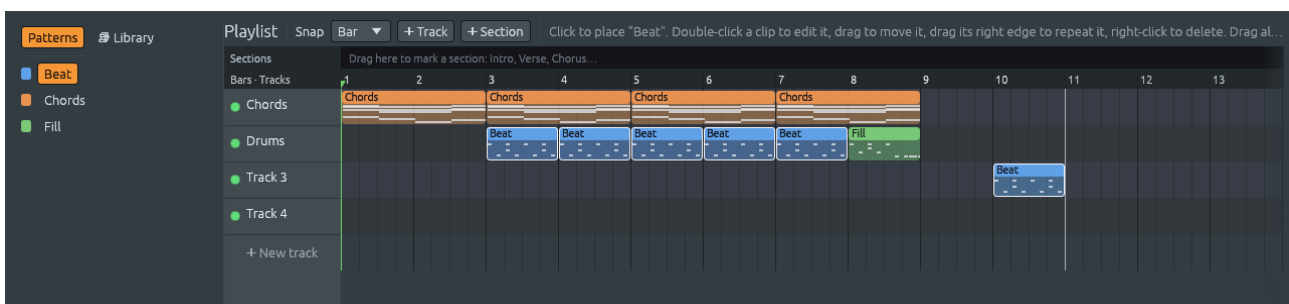
Every closure has its own, unnamed type, so the two branches of the `if` produce values of different types, and a variable can only have one type. `Box<dyn Fn(usize, u8) -> f32>` is the common type: “a closure on the heap that takes a channel and a pitch and returns a height”. `dyn Fn` is a trait object, like `dyn App` in `main.rs`. The `move` keyword makes each closure take ownership of what it uses (`used`, `lowest` ...), which must outlive the `if` that created them.

This runs on the window’s thread, once per clip per frame, so the small allocation for the `Box` does not matter. On the audio thread it would.

A clip’s colour is its pattern’s. Each pattern keeps a `color`, a place in the playlist’s palette, saved in the project file. It does not come from the pattern’s number, because then deleting the first pattern would recolour all the others.

Only the tracks you use

A project has only the tracks down to its last clip: `File > New` has one track, a demo has the ones it uses. Below the last track there is a fainter **spare row**: place a clip on it (or drag one onto it) and it becomes a track. Its header has a **+ New track** button too.



`Project::clip_fits` decides whether a clip may go somewhere. One more row than the tracks is allowed, the one just below the last track, as long as the playlist is not full:

```
fn clip_fits(&self, clip: Clip) -> bool {
    let tracks = self.playlist.tracks.len();
    clip.pattern < self.patterns.len()
        && (clip.track < tracks || (clip.track == tracks && tracks <
MAX_TRACKS))
        && clip.start < MAX_SONG_STEPS
        && (1..=MAX_REPEATS).contains(&clip.repeats)
}
```

`add_clip` and `move_clip` then call `open_track`, which adds a track when the clip is on that new row. Because the track and the clip are added by the same edit, one Ctrl+Z takes both away. The playlist view only has to draw one more row: `Layout` has its `spare` flag and a `rows()` method, and `cell_at` clamps to `rows()`.

Demos are built on a playlist of sixteen tracks, because naming track 0 and track 1 is easier when they exist; at the end, `remove_unused_tracks` drops the empty tracks at the bottom. Files saved before tracks were trimmed need almost the same thing, with one difference: a track someone renamed or muted was kept on purpose, so loading drops only tracks that are exactly as a new track starts. Two rules, one loop, which takes the rule as a closure:

```
pub fn remove_unused_tracks(&mut self) {
    self.trim_tracks(|_, _| true);
}

pub(crate) fn remove_blank_tracks(&mut self) {
    self.trim_tracks(|index, track| *track == Track::numbered(index + 1));
}

fn trim_tracks(&mut self, drop: impl Fn(usize, &Track) -> bool) {
    // ...from the bottom, pop tracks with no clips while `drop` says
    so...
}
```

`impl Fn(usize, &Track) -> bool` means “any function or closure that takes a row number and a track and answers yes or no”. The compiler makes a copy of `trim_tracks` for each closure, so this costs nothing at run time. Comparing with `Track::numbered(index + 1)` works because `Track` derives `PartialEq`: a blank track is equal to a brand new one for its row.

Trimming only ever removes tracks **below** the last clip. An empty track in the middle stays, because removing it would move every clip under it up a row.

Inserting, moving and deleting tracks

A clip names its track by **number**. Insert a track at row 2, and every clip on row 2 or below must move down one, or the drums would suddenly sit on the wrong track:

```
pub fn insert_track(&mut self, index: usize) -> Option<usize> {
    let tracks = &mut self.playlist.tracks;
    if tracks.len() >= MAX_TRACKS {
        return None;
    }
    let index = index.min(tracks.len());
    tracks.insert(index, Track::numbered(tracks.len() + 1));
    for clip in &mut self.playlist.clips {
        if clip.track >= index {
            clip.track += 1;
        }
    }
    Some(index)
}
```

`remove_track` is the mirror image: drop the track's clips with `retain`, then move the ones below up one. `swap_tracks` swaps two rows and the clips on them. Whoever changes the list fixes the numbers that point into it, which is why the fields are private and only `Project`'s methods change them (see [The project](#)). Each of these is one call inside `edit`, so undo brings a deleted track back with its clips.



The menu is egui's `context_menu` on the track's name field. A text field still opens a menu on a right-click, because typing only needs the left button. A track's lower edge can be dragged to make it taller or shorter, and a double-click on it gives it back its usual height.

Following the playhead past the names

With **Follow playhead** on, the playlist scrolls to keep the playhead in view. When a song plays to its end and starts again, or a loop goes back to its start, the playhead jumps left and the view follows it. But the track names on the left are drawn *over* the bars, and to the scroll area they are not there at all: revealing the playhead could leave it under them. So `follow_x` is told how wide the fixed part is, and makes the rectangle it reveals reach that much further left:

```
pub fn follow_x(ui: &Ui, step: Option<usize>, x: f32, view: Rect, covered: f32) {  
    // ...only when the step changed, and Follow playhead is on...  
    let margin = (view.width() * 0.1).min(60.0);  
    let y = view.center().y;  
    ui.scroll_to_rect(  
        Rect::from_min_max(  
            pos2(x - margin - covered, y - 0.5),  
            pos2(x + margin, y + 0.5),  
        ),  
        None,  
    );  
}
```

`scroll_to_rect` scrolls as little as it can so the rectangle is inside the view. Going left, that leaves the rectangle's left edge at the view's left edge, `covered` points left of the playhead, so the playhead ends up right of the names. The playlist passes `HEADER_WIDTH` (its names), the piano roll `KEY_WIDTH` (its keyboard). Going right, nothing changes. Only a new step scrolls, so you can still scroll by hand while the music plays.

Trying the windows

The tools and the project are covered by unit tests, and many of the editors' behaviours by tests that draw the window without a screen (see [Testing a window without a screen](#)). But some problems only show when a person uses the window: a window that creeps taller on every frame, a menu that closes when you pick a choice in it, a chord that lands above the pointer instead of around it. That is a good habit for

any UI code: write the logic so it can be tested without the window, and then still try the window.

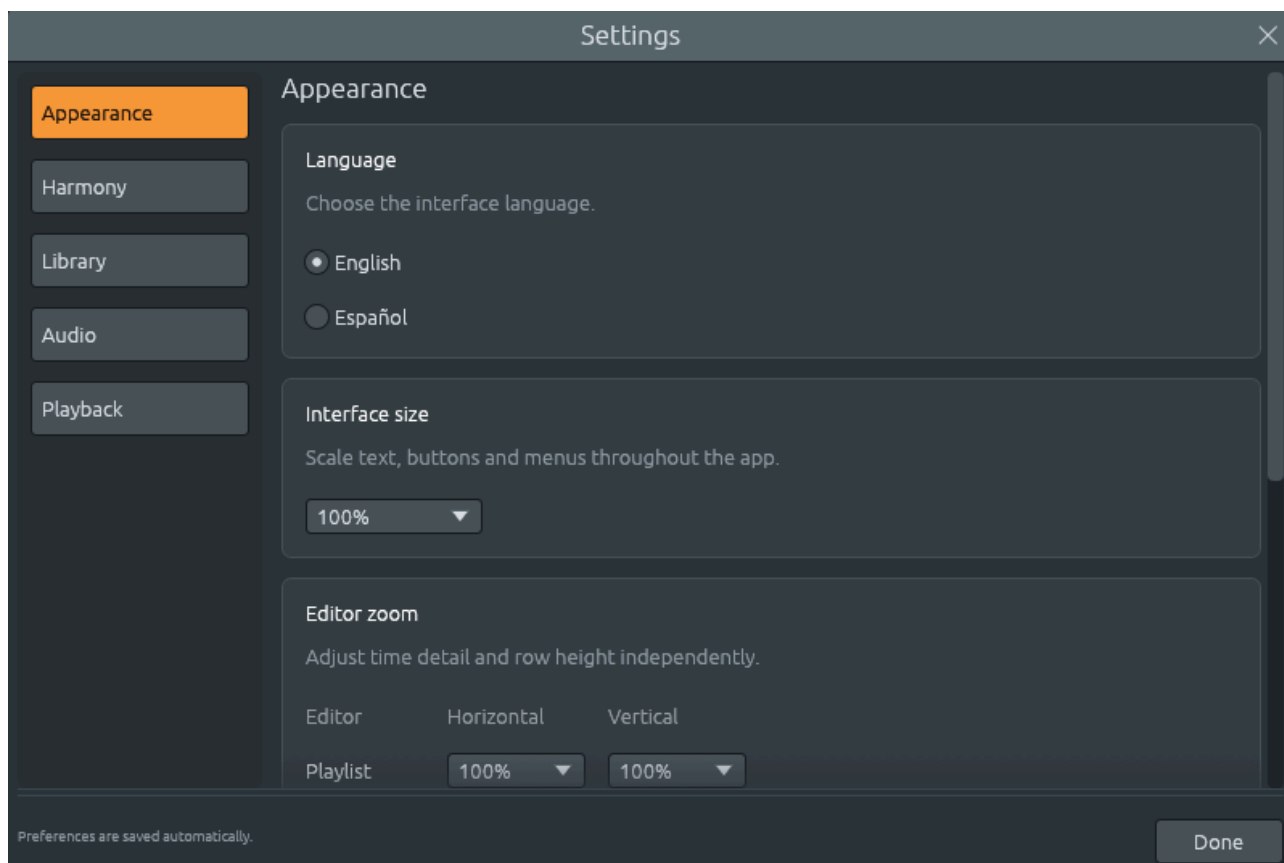
Two languages: translations, and letters or solfège

Files: `crates/lineup-core/src/i18n.rs`, `crates/lineup-core/src/i18n/es.rs`,
`crates/lineup-app/src/i18n.rs`, `crates/lineup-core/src/pitch.rs`,
`crates/lineup-core/src/harmony/symbol.rs`, `crates/lineup-core/src/chord_sheet.rs`, `crates/lineup-core/src/file.rs` and `crates/lineup-app/src/app/settings.rs`

Two settings change how Ferrous Lineup talks to you:

- **Edit > Settings... > Appearance > Language** writes the menus, windows and settings in **English** or **Español**.
- **Edit > Settings... > Harmony > Note naming** names notes, keys and chords with **letters** (C, F#, Bb major, Am7) or in **sofège** (Do, Fa#, Sib major, Lam7).

They are separate, so you can have Spanish menus with letters, or English menus with solfège. Until you choose a note naming, though, it follows the language: Spanish menus name notes in solfège, as Spanish schools do, and English menus with letters. Once you choose one, it stays, whatever language you pick after. Both settings change only what you see. A song saved with one choice opens exactly the same for someone with the other.



Notes named Do, Re, Mi

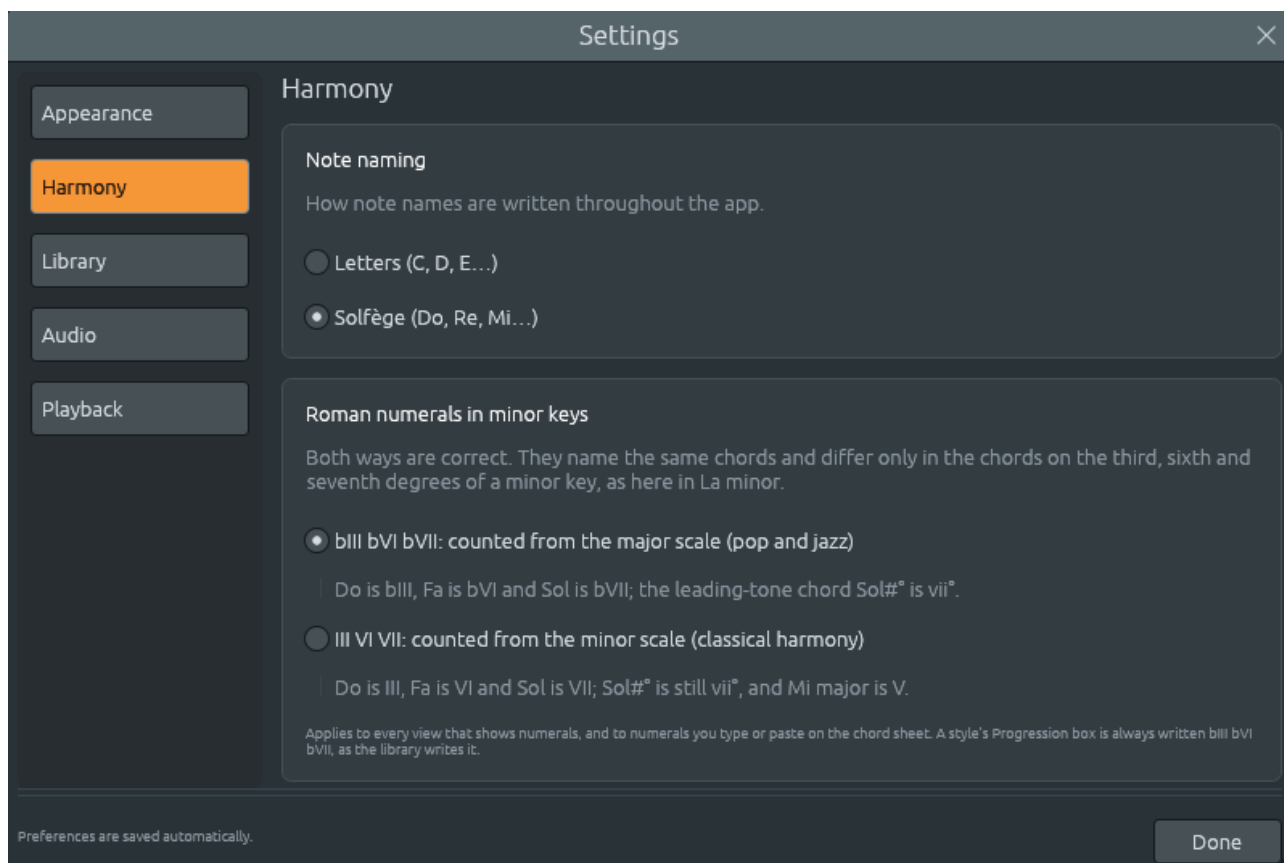
In English and German, notes are named with letters. In Spanish, Italian, French and Portuguese, and in the countries that speak them, they are named with syllables:

letters	C	D	E	F	G	A	B
solfège	Do	Re	Mi	Fa	So ^l	La	Si

Sharps and flats are written after the syllable as after the letter: F# is **Fa#** (*Fa sostenido* in Spanish) and B $\flat$ is **Sib** (*Si bemol*). A key is named the same way, so B $\flat$ major is **Sib major**, or **Sib mayor** with Spanish menus, and D Dorian is **Re dórico**.

This is **fixed do**: Do is always C, whatever the key. Some ear-training methods use a **movable do** instead, where *do* is the tonic of the key (in G major, *do* is G). The app always uses fixed do, as Spanish and Italian schools do, since a note's name must not change with the key.

Octave numbers stay as they were: middle C is C4 in letters and **Do4** in solfège, the octave starting at C and going up to B. Some French and Spanish books number octaves one lower and call middle C Do3. The app keeps one numbering, so a note keeps its number whichever way it is named.

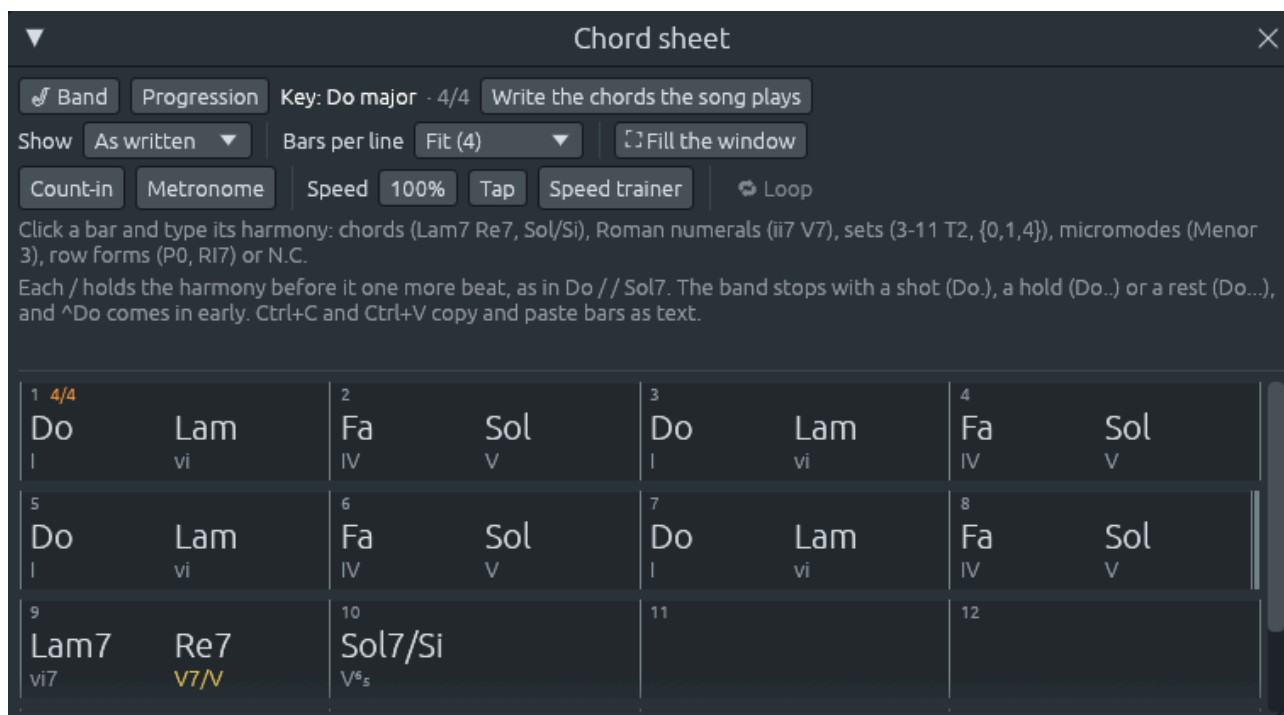


Chords in solfège

A chord symbol is a root, a quality and sometimes a bass note. In solfège only the note names change:

letters	C	Am7	G7/B	Bdim	F#m7b5	Eb+
solfège	Do	Lam7	Sol7/Si	Sidim	Fa#m7b5	Mib+

Spanish-speaking musicians call the first way *cifrado americano* and the second *cifrado latino*. Lead sheets for jazz and pop are usually in letters, even in Spanish; teachers and songbooks often use syllables. With solfège chosen, the chord sheet, the harmony keys, the piano roll, the playlist's key changes and the examples in the hints all write notes, keys and chords the second way.



Typing a chord either way

You can type a chord either way, whichever is chosen: **Am7** and **Lam7** are the same chord. After **Enter**, the bar shows it the way the setting names notes.

A few names could be read both ways. **o** is one way of typing a diminished chord (**Bo** is B°), so **Do** could be C (solfège) or D° (letters), and **Do7** could be C7 or D°7. The app reads such a name the way the setting names notes: with solfège chosen, **Do** is C. Names that only make sense one way are read that way: **Faug** is F augmented even with solfège chosen, because “Fa” followed by “ug” is not a chord, and **Fam** is F minor even with letters chosen.

The quality is written the same in both: F augmented is **Faaug** in solfège. It reads back correctly, but **Fa+** is easier on the eye, and the app reads it too.

On the chord sheet, typing on a selected bar starts typing in it, but **L** is the key that switches between pattern and song. With solfège chosen, La chords start with L, so there L starts typing instead. With letters chosen, no chord starts with L, and L switches as before.

What is saved

Projects, saved styles, the chord-sheet text that **Ctrl+C** copies, and playback traces are always written with letters, and in English. The settings never reach them. So a song you saved while seeing Lam7 contains `Am7`, and opens with Am7 or Lam7 according to the settings of whoever opens it.

In the code

Two settings, kept like the numerals setting

`Language` and `NoteNaming` (both in `lineup-core`) are kept like the [minor-key numerals setting](#): an enum with a `#[default]`, a `thread_local!` `Cell` that every function can read, and a `with` that sets it for a while and puts it back with a `Drop` guard, even after a panic.

```
#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash, Default, Serialize,
Deserialize)]
pub enum NoteNaming {
    #[default]
    Letters,
    Solfege,
}

thread_local! {
    static NOTE_NAMING: Cell<NoteNaming> = const {
        Cell::new(NoteNaming::Letters) };
}
```

A note is printed in hundreds of places, through its `Display` implementation (`{}` in `format!`). A per-thread value lets all of them follow the setting without passing it down through every function:

```
impl fmt::Display for NoteName {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        f.write_str(&self.format(NoteNaming::current()))
    }
}
```

The window sets both values from the saved settings before drawing each frame:

```
pub fn apply_settings(ctx: &egui::Context) {
    language(ctx).set();
    note_naming(ctx).set();
}
```

A video export draws its views on a thread of its own, which starts with the defaults, so it takes the language and the naming along, as it already took the numerals setting.

Solfège until you choose

egui keeps the window's preferences as values under an `Id`, and `get_persisted` gives `None` for an `Id` nothing was stored under. That `None` is what tells “never chosen” apart from a choice, so the naming falls back on the language only then:

```
pub fn note_naming(ctx: &egui::Context) -> NoteNaming {
    ctx.data_mut(|data| data.get_persisted(note_naming_id()))
        .unwrap_or_else(|| language_naming(language(ctx)))
}

/// How notes are named where `language` is spoken.
fn language_naming(language: Language) -> NoteNaming {
    match language {
        Language::English => NoteNaming::Letters,
        Language::Spanish => NoteNaming::Solfege,
    }
}
```

Reading never stores anything; only `set_note_naming`, called when you click a naming in Settings, does. So until that click, every frame asks the language again, and switching to Spanish switches the notes to solfège. The `match` has no `_` arm on purpose: a third language would not compile until someone says how it names notes.

Translating: the English text is the key

Each piece of text the interface shows is written in English in the code and goes through `tr`. It lives in `lineup-core`, not in the app, so that the texts the core writes, the names of styles, instruments, chords and scales, are translated too:

```
pub fn tr(text: &'static str) -> &'static str {
    match Language::current() {
        Language::English => text,
        Language::Spanish => in_spanish(text).unwrap_or_else(||
missed(text)),
    }
}
```

The Spanish catalog is in `i18n/es.rs`: a big `match` for the menus, the main windows and the settings, and, in the files of `i18n/es/`, tables of pairs by area (the app, theory, instruments, the library, the songs), gathered once into a `HashMap`:

```
static TEXTS: LazyLock<HashMap<&'static str, &'static str>> =
    LazyLock::new(|| TABLES.into_iter().flatten().copied().collect());

/// `text` in Spanish, or `None` for a text with no translation yet.
pub fn translate(text: &str) -> Option<&'static str> {
    main_texts(text).or_else(|| TEXTS.get(text).copied())
}
```

`LazyLock` builds the map the first time it is used, on whichever thread asks first, and every thread shares it after that. `flatten` turns the five tables into one run of pairs, and `collect` builds the map from them.

Using the English text itself as the key has two advantages. The code still reads naturally (`tr("Settings")` , not `tr(SETTINGS_TITLE)`), and a text with no translation yet simply stays in English, through `unwrap_or_else` . The cost is that changing an English text means changing its key in the catalog too, or the Spanish one is lost; a test catches that (see below).

Both the argument and the result are `&'static str` : text written in the program itself, which lives as long as the program does. So `tr` can return either one without copying anything.

A few relatives of `tr` cover the other cases:

- `trf("{} steps a bar", &[12])` puts its arguments where the translated text has `{}` , so the Spanish can put the number among other words: “12 pasos por compás”.
- `trc("clef", "Bass")` is for a word whose translation depends on where it is shown: the bass clef is “Fa”, the instrument “Bajo”. The catalog has it as `"clef|Bass"` .
- `tr_name` translates the names the app gives things, a channel, a track or a pattern: “Bass” is “Bajo”, “Pattern 12” is “Patrón 12” and “A drums” is “Batería de A”. A name you typed yourself stays as typed.

- A catalog text can name notes in braces, as in `"{Bb} mayor"`; they are named as the setting says (see [Examples that follow the setting](#)). Such a text is built once for each naming and kept, with `Box::leak`, which turns a `String` into a `&'static str` that is never freed. There are only so many texts and two namings, so the memory kept has a limit.

Names on screen, letters in files

Since `Display` follows the setting, anything that turns a note into text and saves it would save solfège. The places where data leaves the program run with letters instead, through `with`:

```
pub fn to_text(project: &Project) -> Result<String, FileError> {
    // ...
    let file = Language::English.with(|| NoteNaming::Letters.with(||
ProjectFile::from(project)));
    let mut text = NoteNaming::Letters.with(||
ron::ser::to_string_pretty(&file, style));
    // ...
}
```

A chord marker is saved as its text, through the serde attributes `into = "String"` and `try_from = "String"` (see [Twelve-tone rows](#)). Its conversions run with letters too, so a marker is saved as `Am7` wherever it is saved from:

```
impl From<HarmonyName> for String {
    fn from(harmony: HarmonyName) -> Self {
        NoteNaming::Letters.with(|| harmony.to_string())
    }
}
```

Saved styles, the chord-sheet text and the playback trace do the same. The library's own charts are written in letters in the source, so they are read with letters too.

Reading a name either way

`NoteName::read_as` reads a note at the start of a text in one naming, and gives back the name and the rest of the text. For solfège it tries each syllable:

```
NoteNaming::Solfège => Letter::ALL.into_iter().find_map(|letter| {
    text.strip_prefix(letter.solfège())
        .map(|rest| (letter, rest))
})?,
```

`strip_prefix` returns the text after the prefix if the text starts with it, and `None` if not. `find_map` returns the first `Some`. “Sol” and “Si” both start with S, but neither is the start of the other, so only one can match.

`read_either` tries both namings, the setting’s own first, and keeps the first reading whose rest `fits`:

```
pub fn read_either<'a>(
    text: &'a str,
    fits: impl Fn(&'a str) -> bool,
) -> Option<(NoteName, &'a str)> {
    NoteNaming::current()
        .reading_order()
        .into_iter()
        .filter_map(|naming| NoteName::read_as(text, naming))
        .find(|&(_, rest)| fits(rest))
}
```

What “fits” means depends on the caller, so it is a closure. A chord symbol fits when the rest is a quality and maybe a bass; a whole note name fits when nothing is left:

```
pub fn parse(text: &str) -> Option<NoteName> {
    NoteName::read_either(text, str::is_empty).map(|(name, _)| name)
}
```

`str::is_empty` is a function, and a function can be passed wherever a closure with the same arguments is expected.

The lifetime `'a` says that the rest handed to `fits`, and returned, is a piece of `text`, not a copy. That is how “Faug” works with solfège chosen: “Fa” leaves “ug”, which `fits` rejects, so the letters reading, F with “aug”, is tried and kept.

Examples that follow the setting

The hints on the chord sheet and in the settings give example chords. They are written with the notes in braces, `{A}m7 {D}7`, and `with_notes` names each one as the setting says, so with solfège chosen the hint reads “Lam7 Re7”. Braces around anything that is not a note name, like the set `{0,1,4}` in the same hint, are left alone.

L for La

`ChordSheet::takes_l` says whether an L typed now belongs to the sheet: the sheet has the keyboard, a bar is selected, nothing is being typed yet, and notes are named

in solfège. The app's L shortcut checks it first:

```
let l_types = self.chord_sheet.takes_l() && !self.live.typing();
if !typing && !controls_active && !l_types && pressed(egui::Key::L) {
```

`!self.live.typing()` is there because with the typing keyboard on, letters play notes, and L switches pattern and song as usual.

Outside the app

The Linux desktop file has Spanish lines next to the English ones,

`GenericName[es]=Banda de acompañamiento` for example, which a Spanish desktop shows in its menus. The AppStream file that software centres read gives each paragraph of the description a Spanish twin, `<p xml:lang="es">`, and `appstreamcli validate` checks it.

The tests

`every_window_writes_in_spanish` opens six demos in Spanish with every window open, then the shortcuts, About, the video export, the style editor, each page of Settings and each menu. It fails if anything asked `tr` for a text the catalog has no Spanish for, or if any text drawn on screen is an English text the catalog does have Spanish for, which catches text drawn without `tr`. In the core,

`every_name_and_description_the_app_shows_has_spanish` does the same for every instrument, style, chord, scale and demo name the core gives the app.

`solfege_changes_how_names_look_not_what_is_saved` builds every demo song twice, once with letters and once with solfège, and checks that the saved file and the chord-sheet text are identical, and that every chord, written as the sheet shows it in solfège, reads back as the same chord.

`names_are_read_in_letters_or_solfege` reads `Sib7`, `Fa#m`, `Sol` and `Reb`, and checks that `Do7` starts with D when letters are chosen and with C when solfège is.

`solfege_is_a_way_of_showing_names` checks that `Am7` shows as `Lam7`, `G/B` as `Sol/Si` and `Bb` major as `Sib` mayor, and that `Lam7` and `Am7` are the same chord.

`in_solfege_l_types_la_chords_and_in_letters_it_switches_mode` selects a bar on the chord sheet, presses L with solfège chosen and types `Lam7`, then checks that the mode stayed and the bar holds A minor 7; with letters, L switches the mode and leaves the bar alone.

`spanish_names_notes_in_solfege_until_a_naming_is_chosen` switches the language back and forth with no naming chosen, and the naming follows it; then it chooses a naming and switches the language again, and the naming stays.

`spanish_names_notes_in_solfege_and_keeps_letters_once_chosen` does it the way you would: Spanish menus, **Armonía, Letras (C, D, E...)**, and a restart of the app, after which the notes are still letters.

Exercises: the shape of the program

These exercises practise the Rust of this part on the real code. Each one says what is **already there**, so you extend it rather than build it again. Some are small pieces of work with a clear end; the ones called “Think about it” are questions, with no code to write.

Try them on a branch of your own. Run `cargo test` after each change, and `cargo fmt --all --check` and `cargo clippy --workspace --all-targets -- -D warnings` before you call one finished: CI runs all three.

Crates and threads

1. Break it on purpose (any crate)

Learning to read compiler errors is half of learning Rust. Try each of these, read the error, then undo it:

- In `AudioEngine::new ( engine.rs )`, call `build_stream::<f32>(&device, config.clone(), audio_thread, Arc::clone(&problems))` twice in a row, so the second call fails because `audio_thread` was moved into the first.
- Remove `move` from the audio callback closure in `build_stream`.
- Add a variant `SetName(String)` to `Command`. Why can `Command` not be `Copy`? Which variants already make that impossible, and what does the compiler say when you ask for `Clone`?
- Make `Note`'s field `pub` (in `lineup-core/src/pitch.rs`) and write `Note(200)` in `app.rs`. Why does that compile, and why is it a bad idea?

Practises: ownership, `move` closures, `Copy` and non-`Copy` types, privacy.

2. Think about it: the audio thread

- Why would `Command::InsertChannel { instrument: Box<Instrument>, ... }` be worse than sending the instrument by value? (`Instrument` already boxes its voice arrays; what would a second box buy?)
- What would happen if the window forgot to call `collect_garbage`?

- `Garbage` is one enum with a variant for each kind of thing thrown away. What would one trash queue per kind cost, and what would it gain?
- `AudioEngine::playhead` uses `Ordering::Relaxed`. Could the window ever see a step number that was never played?

The project

3. Delete every empty track (lineup-core and lineup-app, easy)

Deleting one track is already there: read `Project::remove_track` and the “Delete track” item in the track’s right-click menu. It removes the track and its clips, and moves the clips of the tracks below it up by one, just as `remove_pattern` does for patterns.

What is missing is a way to remove *all* the empty tracks at once.

`remove_unused_tracks` only drops empty tracks at the bottom of the playlist (it calls `trim_tracks`). Add `remove_empty_tracks`, which removes a track wherever it is if it has no clips, and an item for it in the track menu. Which indices must you visit, and in which order, so that the clips keep pointing at the right tracks? What should happen if that would leave no tracks? Write a test with the demo song first, then the method.

Practises: `retain`, keeping rules that span two lists, testing the order of index changes.

4. Think about it: positions or ids

Clips refer to patterns by their *position* in the list, which is why `remove_pattern` has to renumber them. Imagine instead that every pattern has a permanent id (`struct PatternId(u32)`) that never changes.

- Which methods of `Project` become simpler, and which harder?
- What would change in the file, and in the audio thread’s `ChannelNotes`?
- Would undo be easier with ids or with positions?

Telling the audio thread what changed

5. Random edits for the diff (lineup-core, medium)

The diff tests check hand-picked cases. Write one that checks a thousand random ones: start from the demo, make a random edit (toggle a step, add or remove a pattern or a channel, change a volume, move a clip...), call `check_both_ways(&before, &after)`, and repeat from the new project.

For randomness without any crate, a tiny “xorshift” generator is enough for tests:

```
struct Random(u64);

impl Random {
    fn next(&mut self, below: usize) -> usize {
        self.0 ^= self.0 << 13;
        self.0 ^= self.0 >> 7;
        self.0 ^= self.0 << 17;
        (self.0 % below as u64) as usize
    }
}
```

Start it from a fixed number (not 0, which stays 0 forever), so a failure happens again every time you run the test. If you want to go further, the `proptest` crate does this properly, and when it finds a failing case it *shrinks* it to the smallest list of edits that still fails.

Also try several random edits between two checks, as one undo step might hold. Does `diff` ever produce a long list where a short one would do? Is that a problem?

Practises: property-based testing, reproducible randomness.

6. Channels lined up by name (lineup-core, medium)

`diff` lines up the channels of two projects with `|old, new| old == new`, and the patterns with `same_pattern`. Write `same_channel(old: &Channel, new: &Channel) -> bool`, which says two channels are the same when they have the same name, and pass it to `Alignment::new` instead. Run the diff tests. Then find a case where it gives the wrong answer: what happens with two channels called “Piano”?

Practises: functions as arguments, what “the same” means.

7. Think about it: what to send

- `SetNotes` sends a channel's whole list of notes for every change. At what size would that become a problem, and what would you send instead?
- The window sends the song's length along with its pieces, instead of letting the audio thread work it out. What would it take to keep two copies of that calculation in agreement, and how would you notice if they drifted apart? The pieces are `Play s` made by `Project::playback_arrangement`. Why is unrolling repeats on the window's side simpler than teaching the audio thread about sections?
- Undo while the song plays. When undo removes a channel, its ringing notes stop at once, because the channel goes to the trash. In the case of [three identical Lineup Osc channels](#), the audio thread removes the *last* channel and gives its notes to another one. Which channel's ringing note keeps sounding, and could you ever hear the difference?

Undo and redo

8. An undo history window (lineup-app, easy)

Add Edit > History..., a small window listing every step that can be undone or redone, newest at the top, with the current position marked. Clicking a step jumps there.

You need two methods on `History`: one giving the undo labels and one the redo labels, each as an iterator (`self.undo.iter().map(|step| step.label)`). Jumping back five steps is five calls to `history.undo`. Should `sync_audio` run after each one, or once at the end, comparing with the project from before the first? Which is cheaper, and does the audio thread end up the same either way? (Exercise 5 answers the last question.)

Practises: iterators over private fields, `rev()`, reusing `diff`.

9. Cheaper snapshots (lineup-core, harder)

Every undo step copies the whole project, every pattern included, even though a click on a step changes one pattern. (A drag copies once, when its step opens, not on every frame, but the copy is still the whole project.) Make the copies cheap by sharing what did not change: store the patterns as `Vec<Arc<Pattern>>`.

Cloning a `Vec<Arc<Pattern>>` copies only the pointers and adds one to each count; the patterns themselves are shared between the current project and every copy in the history. To change one, use `Arc::make_mut`, which gives a `&mut Pattern` and first makes a private copy *only if* the pattern is shared (“clone on write”). So

`Project::pattern_mut` becomes

```
self.patterns.get_mut(index).map(Arc::make_mut) .
```

Then `diff` can skip unchanged patterns quickly: if `Arc::ptr_eq` says two patterns are the same memory, they are certainly equal. Make a test project with 999 patterns full of notes, and measure the time of an edit before and after with `std::time::Instant`. Does `PartialEq` for the project still work unchanged? Does the file format need to change?

Practises: `Arc` for sharing, clone on write, measuring before optimizing.

10. Think about it: labels and what undo covers

- Step labels are `&'static str`, so “rename pattern” cannot say which pattern. A `String` label would allow “rename pattern Beat”, but most labels are fixed text, and a `String` would allocate for each of them. Look up `std::borrow::Cow<'static, str>`: how does it let a label be either, and allocate only when it must?
- The master volume is part of the project’s mixer, so undo changes it back, and it is saved with the song. The selected pattern, the zoom and the open windows are not. Would you expect undo to change any of them? What would have to move into `Project` for that, and what would it mean for the saved file?

Saving

11. A new project field and its format test (lineup-core, medium)

Goal. Add one harmless field to a project, save it without breaking old files, and prove that a song which uses it is written as the next format. `ProjectFile`, its `serde` defaults and the file tests are already there; this is the smallest complete exercise in growing the format.

Files. `crates/lineup-core/src/project.rs`, `crates/lineup-core/src/file.rs`.

Steps.

1. Add `pub notes: String` to `Project`, initialized to `String::new()` in `Project::empty`.
2. Add `notes: String` to `ProjectFile`, with `#[serde(default, skip_serializing_if = "String::is_empty")]`, and copy it in both directions.
3. Raise `FORMAT_VERSION` by one, to 30 (if another exercise already took 30, use the next free number). In `format_for`, return the new number first when `!project.notes.is_empty()`; leave everything else as it is.
4. Write three tests in `file.rs`: an empty project stays at its old format and has no `notes:` line; a project with notes is the new format and round-trips; and a file written before the field existed reads back with empty notes.

Run. `cargo test -p lineup-core file`.

Done when. Old project texts still open, the new field survives a round trip, and `format_for` names the new number only for a song that needs it. `files_from_the_future_are_refused` builds its file from `FORMAT_VERSION + 1`, so it keeps passing.

12. A recovery copy that is damaged (lineup-app, easy)

Goal. Make crash recovery trustworthy when the copy on disk is not.

`app/recovery.rs` already writes a project copy, an `.info` file and a lock file for each window to the `recovery` folder in the app's data folder, offers the copy at the next start, and removes a copy that will not load. The test `a_recovery_copy_comes_back_with_its_file` covers the happy path. Add the unhappy ones.

Files. `crates/lineup-app/src/app/recovery.rs`.

Steps.

1. See it by hand: open a demo, type a chord in the first bar, wait a moment, and force-quit the app. Do not use **File > Quit**. Start it again and choose **Recover it**. Close the recovered song without saving it.
2. Read `Files`, `load` and `write`, and the tests at the bottom of the file.
3. Add a test that takes `Files::of_run(&folder, "test")`, creates the folder, writes rubbish into its `project` file, puts a valid `info` file beside it, calls `load(files.clone(), None)`, and checks that it returns `None` **and** that both files are gone.
4. Add a second test that writes a valid project to `files.project` with `file::save`, but leaves out the info file. Check that `load` still returns the project, with no path and no name, so the dialog can call it "Untitled".

5. Remove the temporary folder at the end of each test, as the existing ones do.

Run. `cargo test -p lineup-app recovery`.

Done when. Both tests pass, and the first fails if the `files.remove()` call in `load`'s error branch is deleted.

13. Forget a recent project (lineup-app, medium)

File > Open recent remembers the last ten projects in egui's persisted data, and stays right when a file has been moved or deleted (the open fails and the path is removed). Read `app/recent.rs`.

Extend it: a "Clear recent" item that forgets the whole list, and a small $\times$ next to each entry that forgets one. The $\times$ must not also open the file, so watch the order in which you check the two clicks. Why is the list stored as `Vec<String>` rather than `Vec<PathBuf>`, and what happens to a path that is not UTF-8?

Practises: egui menus, persistence, paths that are not UTF-8.

14. Reading numbers slowly (lineup-core and RON, medium)

The [version number](#) explains why the app no longer reads a file's format through a struct with one field: skipping the rest of the file made RON look ahead, number after number, to the end of the text.

1. Say a saved song is 2 MB of text with 120,000 numbers in it. If skipping each number reads, on average, half the file, how many bytes are read in total? At about 2 GB a second, how long does that take? How long for a song twice as long?
2. Find the look-ahead in RON's source: in the version `Cargo.lock` names, `src/parse.rs` searches `self.src()[skip..]` for `".."`. Rewrite that line so it searches only the first `raw_float_len` characters. (Hint: slice `self.src()[skip..][..raw_float_len]`, and think about what happens if the slice ended in the middle of a character.) Then compare with how RON 0.12.3 does it, and read its comment on why a `..` cannot straddle the end.
3. `format_number` returns `None` for `(tempo: 120.0, format: 25)`. Why is it fine that it only reads the start of the file? (Look at the order of the fields in `ProjectFile`, and at `to_text`.)

Practises: thinking in growth rates, reading a library's source, string slices.

15. Think about it: files

- `has_unsaved_changes` compares the whole project with the saved copy, but only after an edit or an undo sets `check_dirty`, not on every frame. Roughly how many notes would a project need before one comparison takes a noticeable part of a frame (about 16 milliseconds)? Why does `project_edited` set `check_dirty` only when `dirty` is already false? What would a counter that every edit increases add to this design?
- The project file has its own types (`ProjectFile`, `PatternFile`) instead of deriving `Serialize` on `Project` itself. What would go wrong the day someone renames a field of `Project`, if the file used it directly?
- An older Ferrous Lineup refuses to open a file with a newer format number. Could it load what it understands and ignore the rest instead? When would that be helpful, and when would it quietly lose someone's work?
- The damaged-file test writes `NaN` in place of numbers. What happens when a project file says `tempo: NaN`? And `master: NaN`? Find the code that decides each, and say why they differ.

Drawing the windows

16. A double-click resets the volume (lineup-app, easy)

Make a double-click on the toolbar's VOL slider set the master volume back to `lineup_core::Mixer::default().master` (0.8). A Reset button would not fit: the toolbar must stay on one row in a window 1280 points wide, in Spanish too, and `the_toolbar_fits_on_one_row_at_1280_points` checks it.

Call `LineupApp::set_master_volume` (in `app/mixer.rs`) rather than assigning the field, so the change is a step to undo and the diff sends the audio thread `Command::SetMasterVolume`. Run the app and check that the slider moves.

Practises: immediate-mode input, `Response`, routing a change through `edit`.

17. Remember the hint level (lineup-app and lineup-core, medium)

The harmony keys' hint level goes back to Short every time the app starts. Other choices are remembered: [Remembering between runs](#) shows how, and `settings.rs` keeps the minor-numerals choice with `get_persisted` and `insert_persisted` under

an `Id` of its own. Do the same for `HintLevel`: give it `serde's Serialize` and `Deserialize` in `lineup-core` (which already depends on `serde`), and read and write it where `HarmonyKeys` keeps `hints`. Which other choices would you remember the same way?

Practises: `serde` derives across crates, `egui's` persisted data.

18. Hints everywhere (lineup-app, small)

`theme::hint` gives a widget a line in the hint bar and the same text as a tooltip. The toolbar, the harmony keys and some playlist buttons use it; the channel rack and the piano roll still use plain `on_hover_text`. Change them to `theme::hint`, and make the hint bar show the pattern's name when nothing is under the mouse. Why does `theme::hint` take the `Response` by value and give it back, instead of taking `&Response`?

Practises: ownership through a function, `egui's Response`, reading someone else's module before changing it.

19. F10 for the metronome, and its F1 line (lineup-app, easy)

Goal. The metronome is a toggle in the toolbar and in **Edit > Settings... > Playback**. Give it a key and add the key to the F1 list. F10 is free; do not use plain **M**, which plays a note when the typing keyboard is on.

Files. `crates/lineup-app/src/app/practice.rs`, `crates/lineup-app/src/app.rs`, `crates/lineup-app/src/app/shortcuts.rs`, `crates/lineup-app/src/app/ux_tests.rs`.

Steps.

1. In `practice.rs`, add a `KeyboardShortcut` constant for F10, the way `view.rs` keeps F5 to F8.
2. In `LineupApp::keyboard_shortcuts`, read it in the same guarded place as `Home` and `End`: not while a text field has the keyboard, and not while a control is in use. Call `set_metronome` with the opposite of `practice.view().metronome`.
3. Add a line to the **Anywhere** section in `shortcuts.rs`: "Metronome on or off". Use the constant rather than writing the key twice.
4. Add a headless test in `ux_tests.rs`: press F10, assert `practice.view().metronome`; press it again, assert it is off.

Run.

```
cargo test -p lineup-app shortcuts
cargo test -p lineup-app metronome
```

Done when. F10 toggles the toolbar button, F1 lists it under “Anywhere”, and `no_two_global_shortcuts_share_a_key` still passes. A new line in the F1 list is English text: the test `every_window_writes_in_spanish` tells you it needs a Spanish one too (see [Two languages](#)).

20. A picture and a version (lineup-app, easy)

- Draw another picture for **Follow playhead** in `icons.rs`: a playhead with two arrows. Which test tells you when a line goes outside its square?
- Add a **Copy version** button to the About window that puts “Ferrous Lineup” and the real version on the clipboard with `ctx.copy_text`. Can the text be a `const`, built with `concat!` and `env!`?

Practises: painting with egui, compile-time strings.

21. Think about it: lists and windows

- In `shortcuts.rs`, the “Harmony keys” section uses `Written("1 to 7")`. Why not seven `KeyboardShortcut`s in an `Any`? What would the list show then?
- The lead sheet opens with `Opening::FirstTime`. What would change for you if it used `EveryTime`? Try it, and say which you prefer.

The editors

22. Clear a row (lineup-app, easy)

Right-clicking one step already clears it (`RackAction::ClearStep`). Add “Clear steps” to a channel’s name menu in `channel_menu`, with a new `RackAction::ClearRow { channel }`. In `LineupApp::apply`, call `edit_pattern` to switch every step of that channel off, without sending the audio thread a command by hand: the edit’s diff sends the new notes. Why is one whole-row edit better than a loop of single-step edits?

Practises: the action pattern, keeping project and audio in step.

23. Paint steps by dragging (lineup-app, medium)

In the rack you click one step at a time. Give `widgets::step_button` a `Sense::click_and_drag()` and make `Rows::steps` paint every step the pointer passes over. `Rows` is rebuilt every frame and only returns actions, so keep the first step, and whether it was being switched on or off, in `LineupApp` (or in egui's memory) and pass it in.

Add `RackAction::SetStep { channel, step, on }` rather than reusing `ToggleStep`, so a step the pointer crosses twice does not flip back. Check `response.hovered()` together with `ui.input(|i| i.pointer.primary_down())`, and stop painting when the pointer leaves the row. Should the whole drag be one step to undo? See [When does one step end?](#)

Practises: immediate-mode state, egui input, the action pattern.

24. Show the scale degree on the keys (lineup-app, easy)

`paint_keys` in `piano_roll.rs` draws a dot on every key of the scale. Draw the degree number instead (1 for the root, 2, 3...), using `Scale::degree_of`. A twelve-tone row already numbers its notes; keep that path as it is. Musicians use these numbers all the time ("the 5 chord", "a flat 7").

Practises: `Option` in drawing code, `painter.text`.

25. The rack's lights (lineup-app, easy)

1. Make the lit light green instead of orange. Where is the colour defined, and what else uses it?
2. Give the lights on the first step of each bar a colour of their own, instead of only a brighter one.
3. While the song plays a different pattern than the rack shows, the lights stay dark. Add a faint note in the rack, such as "Not playing now", for that case. Which values in `channel_rack::show` tell you the song is playing?

Practises: reading drawing code, egui colours.

26. Shorter chord names (lineup-app, medium)

The piano roll's chord lane leaves out a name that does not fit its block. Make it try a shorter name first: for a slash chord, the chord without its bass ("E7" for "E7/B"), and then the root alone ("E"). Lay out each with `layout_no_wrap` and draw the first one that fits. Is a chord without its bass still the right name for what plays?

Practises: measuring text before drawing it, iterators with `find`.

27. Alt-drag to copy a clip (lineup-app, easy)

Ctrl+D duplicates the selected clips, but dragging a copy is quicker when you want it somewhere else. Shift is taken (Shift-click adds a clip to the selection), so use Alt.

In `PlaylistView::handle_input`, read the modifier keys with `ui.input(|input| input.modifiers.alt)`. The obvious plan is to add a copy on the press and drag the copy. But `add_clip` refuses a clip of the same pattern that already starts at the same place, so a copy cannot be added on top of the original. Instead, remember in `ClipDrag` that this drag copies, and add the copy only when the pointer reaches another cell. Which index does the new clip get in `Project::playlist().clips()`, and how does the drag learn it?

Practises: reading egui input, reusing actions you already have.

28. A pattern colour picker (lineup-app, medium)

Pattern colours are already saved. `Pattern` has a public `color` (an index into `PALETTE` in `playlist.rs`), `PatternFile` writes it when it is not the pattern's place in the list, and files without it load with the pattern's place as its colour. Read `PatternFile` and `pattern_color`.

What is missing is the picker. Add one to the ... pattern menu in the toolbar, with a little square for each palette colour, that sets the pattern's colour through `edit_pattern`. Then add "Reset colours", which gives every pattern its place in the list again. Does changing a colour need a new `FORMAT_VERSION`? What does an older Ferrous Lineup do with a file that has one?

Practises: evolving a file format, `#[serde(default)]`, routing a change through `edit`.

29. Go to a bar (lineup-app, medium)

Home, End, Page Up and Page Down already move the playhead: read the transport keys in `LineupApp::keyboard_shortcuts`, `seek` and `skip_bars`. Add a small “bar” box next to the position display, where typing a bar number and pressing Enter goes there. A bar’s first step depends on every meter change before it: which function already turns a step into a bar, and can it go the other way? While you type in the box, Home and End must move the text cursor, not the playhead: what keeps them from doing both?

Practises: keyboard focus, converting bars to steps.

30. Think about it: what a drag remembers

The single-note drag in the piano roll remembers a note by its index in the list (`Drag::Move { index, .. }`), while a selection remembers the notes themselves. What could go wrong with the index if the list were kept sorted by start time, and a note was moved while being dragged?

Two languages

31. Letters and solfège (lineup-core and the app)

1. With solfège chosen, type `Do` in an empty bar of the chord sheet. Then choose letters and type `Do` in the next bar. What does each bar hold, and why are they different?
2. Write the seven chords of Sib major in solfège, then check them on the harmony keys. Which chord is diminished?
3. A project is saved with letters more than once over: `file::to_text` does its work inside `NoteNaming::Letters.with`, and so does `From<HarmonyName> for String`. Remove every `NoteNaming::Letters.with` from both, keeping what is inside, and run `cargo test -p lineup-core solfege`. Which check fails, and what would a friend using letters find in your file? Then put back only the one in `From<HarmonyName>`. The tests pass again: why, and why keep the others too?
4. `with_notes` leaves `{0,1,4}` alone. Find the line in `with_notes` that decides this, and say what it would do with `{Do}`.
5. `note_naming` could store the language’s naming the first time it reads none, with `insert_persisted`, so the next read finds it. It would look tidier. What would

then go wrong for someone who starts the app in English and switches it to Spanish a week later?

Practises: thread-local settings, string slices, reading tests.

The sequencer: timing to the sample

Files: `crates/lineup-core/src/sequencer.rs` and `crates/lineup-audio/src/engine.rs`

The audio thread plays a pattern or the song by counting audio frames. The counting is done by a small type in `lineup-core`, the **sequencer**, which knows nothing about sound: it only says on which frame each step starts. Everything that sounds on a step, the notes of every channel in every clip, is the audio thread's side, in `engine.rs`. This chapter follows a step from the frame count to the notes it starts. How the audio thread itself runs is in [Two threads](#); how tempo changes, meters and swing bend the steps is in [Tempo, meter, swing and feel](#).

The arithmetic

A step is a sixteenth note, and a beat a quarter note, so a beat has four steps (`STEPS_PER_BEAT`). At a tempo of T beats per minute and a sample rate of R frames per second, one beat lasts $R \times 60 / T$ frames and one step a quarter of that:

```
frames per step =  $R \times 60 / T / 4$ 
```

```
120 BPM, 48 kHz:  $48000 \times 60 / 120 / 4 = 6000$  frames      (exact)
130 BPM, 44.1 kHz:  $44100 \times 60 / 130 / 4 = 5088.46\dots$  frames (not whole!)
```

```
pub fn frames_per_step(tempo: Tempo, sample_rate: u32) -> f64 {
    tempo.samples_per_beat(sample_rate) / STEPS_PER_BEAT as f64
}
```

If 5088.46 were rounded to 5088, every step would be 0.46 frames short. That is about 10 microseconds, nothing you would hear once, but it adds up: after a few minutes the drums would be audibly ahead of anything else running at 130 BPM. So the sequencer keeps the fraction, in an `f64`.

Ticking

The audio thread calls `tick` once for every frame it renders:

```
pub fn tick(&mut self) -> Option<usize> {
    let fired = if self.frames_until_next <= 0.0 {
        let step = self.next_step;
        if self.coming_round {
            self.laps += 1;
        }
        self.next_step = (self.next_step + 1) % self.length;
        self.coming_round = self.next_step == 0;
        self.frames_until_next += self.frames_per_step;
        Some(step)
    } else {
        None
    };
    self.frames_until_next -= 1.0;
    fired
}
```

`frames_until_next` counts down by one each frame. When it reaches zero, the next step fires and a whole step's length is *added* to the counter, not assigned. Whatever fraction was left over stays in the counter, so step boundaries land at 0, 5088.46, 10176.92... rounded to the nearest frame each time, but never drifting. `%` is the remainder operator: it wraps the step number back to 0 at the end of the loop.

Because a step starts on the exact frame where it begins, not at the start of the audio block that contains it, the timing does not depend on the sound card's buffer size. This is what **sample-accurate** means.

What else the sequencer can tell

Other parts of the program ask the sequencer more than “which step now”:

- **position** gives the step with the fraction that has passed: 4.5 is halfway through step 4. Recording uses it to put a key played between two steps on the nearer one ([MIDI](#)).
- **laps** counts how many times the loop came round by playing on past its last step. A jump back to the start does not count. The speed trainer goes faster on each lap ([Practice tools](#)).
- **due** says which step starts on the *next* tick, without moving. The audio thread asks it before `tick`, so a song whose tempo changes can set the new tempo before that step starts, and the whole step lasts as long as it should.
- **start_at** jumps, so the given step fires on the next tick; a step past the end wraps round, as playback would.

`set_tempo` changes the length of the steps to come; the step already playing keeps the frames it has left. `set_length` wraps to step 0 if the next step is past the new end.

Why it lives in `lineup-core`

The sequencer has nothing to do with sound cards. It only counts. Keeping it in `lineup-core` means it is tested in milliseconds, without audio, and the count-in uses a second one of its own.

Testing with iterators

```
fn run(sequencer: &mut Sequencer, frames: usize) -> Vec<(usize, usize)> {
    (0..frames)
        .filter_map(|frame| sequencer.tick().map(|step| (frame, step)))
        .collect()
}
```

This helper ticks the sequencer `frames` times and lists every step that fired together with the frame it fired on:

- `0..frames` is a **range**, which is an iterator over numbers.
- `filter_map` calls the closure for each frame and keeps only the `Some` results, unwrapped.
- `Option::map` turns `Some(step)` into `Some((frame, step))`, a tuple.
- `collect()` gathers the results into a `Vec`. Rust works out which collection to build from the return type.

The tests then check exact frame numbers:

```
assert_eq!(
    fired,
    vec![(0, 0), (6_000, 1), (12_000, 2), (18_000, 3), (24_000, 0)]
);
```

and that one minute at 130 BPM gives exactly 520 steps (130 beats × 4), the last within one frame of where 519 exact steps put it. Underscores in number literals like `6_000` are ignored by the compiler and just make big numbers easier to read.

Pattern mode and song mode

```
#[derive(Debug, Clone, Copy, PartialEq, Eq, Default)]
pub enum PlayMode {
    /// Loop the selected pattern, to work on it.
    #[default]
    Pattern,
    /// Play the playlist from the start, and loop at the end of the song.
    Song,
}
```

The sequencer is the same in both modes. It counts steps from 0 to some length and loops. Only the length changes: the selected pattern's length in pattern mode, the song's length in song mode.

```
fn update_loop_length(&mut self) {
    let length = match self.mode {
        PlayMode::Pattern => self.pattern_length(self.pattern),
        PlayMode::Song => self.song_length,
    };
    let length = if length == 0 { STEPS_PER_BAR } else { length };
    self.sequencer.set_length(length);
}
```

An empty song would have length 0, and a loop of zero steps makes no sense, so it becomes one silent bar.

The song arrives as `Command::SetSong { plays, length }`: the pieces of patterns to play, with every repeat already unrolled, and the length. The audio thread could work the length out, but `Project` already does, and two copies of one calculation can drift apart. What a `Play` is, and how the project builds them, is in [Sections, repeats and loops](#).

Notes for every pattern: ChannelNotes

Each channel needs one list of notes per pattern. A `Vec` of `Vec`s does that, but there is a catch. When a pattern is added, the audio thread must add an empty list to every channel. `Vec::new()` itself is free: an empty `Vec` has no memory of its own yet. But adding it to the *outer* `Vec` allocates if the outer `Vec` is full. To be safe, the outer `Vec` must have room for every pattern there could ever be, reserved before it reaches the audio thread.

That is a rule about how the value was made, and Rust can enforce such rules with a **newtype**, a struct with one private field:

```
pub struct ChannelNotes(Vec<Vec<NoteEvent>>);

impl ChannelNotes {
    /// One list per pattern, in pattern order.
    pub fn new(lists: impl IntoIterator<Item = Vec<NoteEvent>>) -> Self {
        let mut patterns = Vec::with_capacity(MAX_PATTERNS);
        patterns.extend(lists.into_iter().take(MAX_PATTERNS));
        ChannelNotes(patterns)
    }

    /// Adds an empty list for a new pattern at position `pattern`,
    without
    /// allocating. The lists from there on move up one place.
    fn insert_empty(&mut self, pattern: usize) {
        if self.0.len() < self.0.capacity() {
            self.0.insert(pattern.min(self.0.len()), Vec::new());
        }
    }
}
```

Because the field is private, code outside `engine.rs` cannot write `ChannelNotes(some_vec)`. The only way to make one is `ChannelNotes::new`, which always reserves room for `MAX_PATTERNS` lists. So every `ChannelNotes` the audio thread receives has the room, and `insert_empty` can never allocate. The check `len < capacity` makes extra sure: if it were ever full, it would rather skip the insert than allocate. `.0` is how you reach the field of a tuple struct (one whose fields have no names). The test `channel_notes_have_room_for_every_pattern` pins the promise down.

On the window's side, sending a channel builds one with an iterator, one list per pattern:

```
// The notes as they are played, with the song's swing.
let notes = ChannelNotes::new(
    (0..self.project.patterns().len())
    .map(|pattern| self.project.played_notes(pattern,
index).into_owned()),
);
```

`new` takes `impl IntoIterator<Item = Vec<NoteEvent>>`, so it accepts this iterator, an array like `[steps(&[0]), steps(&[2])]` in the tests, or a `Vec` of lists. The notes are `played_notes`, not the written ones: with swing, the second eighth of each beat comes late ([Tempo](#), [meter](#), [swing and feel](#)).

Patterns come and go

`RemovePattern` takes the pattern's list out of every channel. The lists after it move down one place, exactly as `Project::remove_pattern` renumbers the clips, so both sides keep agreeing about what "pattern 2" means. Each removed list goes to the trash queue, to be freed on the window's thread:

```
fn remove_pattern(&mut self, pattern: usize) {
    if pattern < self.pattern_lengths.len() {
        self.pattern_lengths.remove(pattern);
        for index in 0..self.channels.len() {
            if let Some(notes) =
self.channels[index].notes.remove(pattern) {
                self.throw_away(Garbage::Notes(notes));
            }
        }
        self.update_loop_length();
        ...
    }
}
```

The loop counts with an index instead of `for channel in &mut self.channels`, because the body calls `self.throw_away`, which needs `&mut self`. A `for` loop over `&mut self.channels` would hold a mutable borrow of part of `self` for the whole loop, and `self.throw_away` would need another. With an index, each `self.channels[index]` borrow ends on the same line.

`Clear` empties everything before a new project is sent, when a file is opened:

```
while let Some(mut channel) = self.channels.pop() {
    silence(&mut channel.instrument, &mut self.midi_out, &mut
self.midi_backlog);
    self.throw_away(Garbage::Channel(channel));
}
// `clear` empties the list but keeps its memory: nothing is freed.
self.pattern_lengths.clear();
let old = std::mem::take(&mut self.plays);
self.throw_away(Garbage::Song(old));
```

- `pop` takes the last item out of a `Vec`, or returns `None` when it is empty, so `while let Some(..)` runs until nothing is left.
- `silence` tells an external synth to let go of the notes a MIDI Out channel is holding, since a synth only stops a note when it is told to.
- `clear` on a `Vec<usize>` sets its length to zero but keeps its memory, so nothing is freed here, and later pushes do not allocate.

- `std::mem::take` takes the value out and leaves the type's default in its place, here an empty `Vec` (which owns no memory).

Opening a project throws away up to `MAX_CHANNELS` (32) channels, each with its instrument and notes, and the song, its tempos and its metronome clicks, which is why the trash queue has 1024 places.

Starting the notes of a step

When a step fires, the audio thread starts the notes that begin on it. In pattern mode that is one pattern. In song mode, a step of the song can be inside several clips at once, one per track, and each clip has its own idea of “which step”:

```
PlayMode::Song => {
    // `self.plays` is read while `self.channels` is changed: two
    // different fields, so the borrows do not overlap.
    for play in &self.plays {
        let length =
self.pattern_lengths.get(play.pattern).copied().unwrap_or(0);
        if let Some(local_step) = play.pattern_step(step, length) {
            // Notes stop at the end of the pattern, or of the
            // piece, whichever comes first.
            let room = (length - local_step).min(play.end() - step);
            start_pattern_notes(
                &mut self.channels,
                Origin {
                    pattern: play.pattern,
                    pattern_step: local_step,
                    song_step: step,
                    over_song: false,
                },
                room,
                timing,
                &mut self.pending,
                self.trace.as_deref_mut(),
            );
        }
    }
    ...
}
```

A clip that starts on song step 32 plays its pattern's step 2 on song step 34. If two clips overlap (a Chords clip and a Beat clip, on two tracks), both start their notes. A clip that repeats its pattern is one long `Play`, and `pattern_step` takes the song step modulo the pattern length, so the pattern comes round again and again inside it.

Borrowing only what you need

`start_pattern_notes` is a plain function, not a method:

```
fn start_pattern_notes(
    channels: &mut [ChannelState],
    origin: Origin,
    room: usize,
    timing: Timing,
    pending: &mut Vec<Pending>,
    mut trace: Option<&mut Tracer>,
) { ... }
```

If it were a method taking `&mut self`, calling it inside `for play in &self.plays` would not compile: the loop borrows `self.plays` to read, and the call would borrow *all* of `self` to change. As a function taking `&mut self.channels`, it borrows only that field, which does not overlap with `self.plays`. Shaping a function's arguments so it borrows only what it needs is a common way to work with the borrow checker rather than against it.

`Origin` says where the notes come from: the pattern, the step of the pattern, the step of the song, and whether they come from a pattern played over the song (below). Every voice keeps the origin of the note it plays, which is how the audio thread later knows whether the song still plays it.

`Timing` bundles the numbers that always travel together:

```
/// How long a step is, the sample rate the instruments run at, how
/// loose the timing is, and the frame the step starts on.
#[derive(Debug, Clone, Copy)]
struct Timing {
    step_frames: f64,
    sample_rate: u32,
    loose_frames: f64,
    now: u64,
}
```

Without it, the function would take nine arguments, several of them plain numbers, and a call with four numbers in a row is easy to get in the wrong order. (Clippy's `too_many_arguments` lint complains from eight, and readability suffers before that.)

How long a note is held

A note's length is in steps; the instrument wants frames. The audio thread works it out when the note starts, and calls it the **gate**:


```

let tick = timing.step_frames / TICKS_PER_STEP as f64;
let held = if note.length <= room {
    note.length as f64 * timing.step_frames - f64::from(note.nudge +
note.trim) * tick
} else {
    room as f64 * timing.step_frames - f64::from(note.nudge) * tick
};
let gate = held as u32;

```

A note that runs past the end of its pattern, or of the piece of the song that plays it, is cut there (`room`), so a pattern's last chord does not hang over the next clip. `nudge` and `trim` are fractions of a step, in ticks: a note from a MIDI performance can start a little after its step and end a little before its last one. With loose timing, each note also starts a little late on purpose. A note that starts late waits in `pending`, a list reserved in advance, until its frame comes. Both are in [Tempo, meter, swing and feel](#).

When the song changes under a playing note

The song can change while it plays: a clip is moved or deleted, a track is muted, the band writes new patterns. A note that already started would ring on to the end of its gate, even though the song no longer plays it. So when a new song arrives while playing, the audio thread lets go of the notes it no longer plays:

```

fn release_notes_left_out(&mut self) {
    // The fields are borrowed one by one, so the closure can read the
    // song while the channels are changed.
    let plays = &self.plays;
    let lengths = &self.pattern_lengths;
    let still_played = |origin: Origin| {
        if origin.over_song {
            return true;
        }
        let length = lengths.get(origin.pattern).copied().unwrap_or(0);
        plays.iter().any(|play| {
            play.pattern == origin.pattern
            && play.pattern_step(origin.song_step, length) ==
Some(origin.pattern_step)
        })
    };
    for channel in &mut self.channels {
        channel.instrument.release_unless(still_played);
    }
    ...
}

```

A note is still played if some piece of the new song plays the same step of the same pattern on the same song step. `release_unless` takes the test as a closure (`impl Fn(Origin) -> bool`), so each kind of instrument can let go in its own way: a synth voice goes to its release, a sampler fades its sound out quickly, a MIDI Out sends a note-off. Keys held on a keyboard and previews have no origin, so they are never cut. `notes_stop_when_the_song_no_longer_plays_them` checks it.

The closure borrows `self.plays` and `self.pattern_lengths`, and the loop borrows `self.channels` to change it. That is allowed because they are different fields: since the 2021 edition, a closure that uses `self.plays` captures only that field, not all of `self`, so writing `self.plays` inside the closure would work as well. The two local references only make plain which parts of `self` it reads.

Seeking, and switching mode

`Command::Seek(step)` moves the start position. While stopped, that is where the next `Play` starts. While playing, playback jumps there at once:

```
fn seek(&mut self, step: usize) {
    self.start = step;
    self.pending.clear();
    ...
    if self.playing {
        // Notes that were playing would otherwise hang on
        // until the end they will now never reach.
        for channel in &mut self.channels {
            channel.instrument.release_all();
        }
        ...
        self.sequencer.start_at(step);
    }
}
```

Notes waiting in `pending` belong to the old position, so they are dropped. Switching between pattern and song mode starts the new loop from its top, and releases held notes for the same reason. Stopping releases every note, keys held down included: each one goes into its release, so the sound rings out a moment and does not hang on.

A pattern over the song

The channel rack's ► **Play over the song** plays the selected pattern over the song as it plays, to try it out with the rest of the band (`Command::SetOverlay`). It starts on the song's next bar line, so it comes in in time, and loops for as long as it plays:

```
fn overlay_step(&mut self, step: usize) -> Option<(usize, usize)> {
    let bar_line = click_at(&self.song_clicks, step) == Click::Bar;
    let overlay = self.overlay.as_mut()?;
    // After a jump it waits for a bar line, to start again from its
    // top in time with the bars.
    if overlay.last.is_some_and(|last| step != last + 1)
        && let Some((pattern, _)) = overlay.playing.take()
    {
        overlay.coming.get_or_insert(pattern);
    }
    overlay.last = Some(step);
    if bar_line && let Some(pattern) = overlay.coming.take() {
        overlay.playing = Some((pattern, 0));
    }
    let (pattern, at) = overlay.playing?;
    let length = self.pattern_lengths.get(pattern).copied().unwrap_or(0);
    if length == 0 {
        return None;
    }
    let at = at % length;
    overlay.playing = Some((pattern, at + 1));
    Some((pattern, at))
}
```

The bar lines come from the song's metronome clicks, which already say where each bar starts in every meter. A pattern chosen while another plays over the song waits as `coming` until the next bar line. A jump (a seek, or the song coming back round) is noticed because the step is not the one after the last, and the pattern starts again from its top on the next bar line.

The window draws a second playhead in the rack for the pattern played over the song. Two numbers, the pattern and its step, must be read together, or the window could see a new pattern with an old step. So they share one `AtomicU64`, the pattern in the high 32 bits and the step in the low 32:

```
fn store_overlay(overlay: &AtomicU64, playing: Option<(usize, usize)>) {
    let bits = playing.map_or(NO_POSITION, |(pattern, step)| {
        ((pattern as u64) << 32) | step as u64
    });
    overlay.store(bits, Ordering::Relaxed);
}
```

`<<` shifts the pattern's bits up 32 places, and `|` (bitwise or) puts the step in the empty low half. Reading undoes it with `>> 32` and a mask, `bits & u64::from(u32::MAX)`. One load, one store: an atomic integer cannot be read half old and half new. A special value, `NO_POSITION`, means “nothing plays over the song”.

Testing it

The tests run the audio thread without a sound card (see [Testing without a sound card](#)). A small helper finds where sounds start in a block of samples:

```
fn onsets(out: &[f32]) -> Vec<usize> {
    (0..out.len())
        .filter(|&i| out[i] != 0.0 && (i == 0 || out[i - 1] == 0.0))
        .collect()
}
```

The test click is a short burst after silence, so every hit is an onset. Then a song can be checked in one line:

```
#[test]
fn song_mode_plays_the_clips() {
    let mut h = two_patterns();
    // Pattern 1 at bar 1, pattern 0 at bar 2: a two-bar song.
    let plays = vec![whole(1, 0), whole(0, 16)];
    send(&mut h, Command::SetSong { plays, length: 32 });
    send(&mut h, Command::SetMode(PlayMode::Song));
    send(&mut h, Command::Play);
    let out = render(&mut h, 40 * STEP);
    // Step 2 of the first clip, step 0 of the second, then the song
    // loops back to the start: step 2 of the first clip again.
    assert_eq!(onsets(&out), vec![2 * STEP, 16 * STEP, 34 * STEP]);
    assert_eq!(
        h.playhead.load(Ordering::Relaxed),
        7,
        "step 39 is step 7 of the second time round"
    );
}
```

The last assertion carries a message saying *why* 7 is right: after 40 steps of a 32-step song, the sequencer is on its second time round. A reader who expects 39 sees the reason as soon as the test fails.

Other tests cover the rest of this chapter: clips that overlap, pieces that repeat their pattern, patterns inserted and removed under the others, `clear` sending everything

to the trash, **Play** starting where a seek put it, and seven tests on a pattern played over the song.

Tempo, meter, swing and feel

Files: `crates/lineup-core/src/meter.rs`, `crates/lineup-core/src/tempo.rs`, `crates/lineup-core/src/remeter.rs`, `crates/lineup-core/src/swing.rs`, `crates/lineup-audio/src/engine.rs`, `crates/lineup-app/src/playlist/tempo.rs`, `crates/lineup-app/src/time_picker.rs` and `crates/lineup-app/src/app/swing.rs`

In Ferrous Lineup a step is always a sixteenth note. A waltz, a ritardando, a shuffle and a Bulgarian 7/8 are all written on that same grid. What changes is how the steps are *counted* and *played*:

What	What it changes	Where
A time signature	Where bar and beat lines fall; never a note	<code>meter.rs</code>
A tempo change	How long each step lasts	<code>tempo.rs</code> , <code>engine.rs</code>
Putting the song in another meter	Every note, so each bar becomes a bar of the new meter	<code>remeter.rs</code>
Swing	When the second eighth of each beat is played, not where it is written	<code>swing.rs</code>
Loose timing	How late each note starts, as it plays	<code>engine.rs</code>

The music behind all of this (beats and bars, two and three, odd meters, what a ritardando and a swing are) is explained in the user manual: [Rhythm and meter](#), [Tempo, time signatures and key changes](#), [Swing and feel](#) and [Odd meters and the Balkans](#). This chapter is about the code.

TimeSignature: two numbers and a grouping

```
#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash, Serialize, Deserialize)]
#[serde(from = "SavedTime", into = "SavedTime")]
pub struct TimeSignature {
    /// How many notes of `unit` length make a bar: the top number.
    pub beats: u8,
    /// Which note counts as a beat: 4 for quarter notes, 8 for eighths.
    /// The bottom number.
    pub unit: u8,
    /// Where the felt beats start, when they are not the usual ones: bit
    /// `i` is set when a beat starts on the bar's `i`-th `unit` note.
    /// 0 means the usual beats (see [TimeSignature::usual_groups]).
    ///
    /// A `u64` keeps `TimeSignature` `Copy`, so it can be passed around
    /// as freely as a number; a `Vec` could not be.
    groups: u64,
}
```

The top number can be anything from 1 up to what fits in a pattern; the bottom one is 1, 2, 4, 8 or 16 (`UNITS`). There is no 32: a bar of 7/32 would end in the middle of a sixteenth-note step.

Everything else is worked out from those numbers, and the arithmetic is written so that a damaged file can't make it divide by zero:

```
pub fn unit_steps(self) -> usize {
    (16 / usize::from(self.unit.clamp(1, 16))).max(1)
}

pub fn steps_per_bar(self) -> usize {
    let unit = usize::from(self.unit.clamp(1, 16));
    (usize::from(self.beats) * 16 / unit).max(1)
}
```

`clamp` keeps the unit between 1 and 16, so the division always has something to divide by, and `max(1)` keeps a bar at least one step long, because “which bar is step 50 in?” divides by the bar's length. The menus and `set_time_marker` only accept time signatures that pass `is_valid`, but a hand-edited file can still say `beats: 0` or `unit: 0`, and the test `nonsense_does_not_divide_by_zero` checks that the arithmetic survives it.

`new` is a `const fn`, so time signatures can be constants:

```
pub const COMMON: TimeSignature = TimeSignature::new(4, 4);

pub const ALL: [TimeSignature; 10] = [
    TimeSignature::new(2, 4),
    TimeSignature::new(3, 4),
    // ...
];
```

A grouping inside a Copy type

In 7/8 the seven eighths are felt as beats of two and three, and the order matters: 2+2+3/8 is a Bulgarian rachenitsa, 3+2+2/8 a Greek kalamatianos. The grouping is part of the time signature.

`TimeSignature` is passed by value everywhere, which needs it to be `Copy`: copied like an integer, with no heap memory. A grouping would naturally be a `Vec<u8>`, `[2, 2, 3]`, but a type that holds a `Vec` can't be `Copy`. So the grouping is a **bit mask** instead: bit `i` is set when a beat starts on the bar's `i`-th note. 2+2+3/8 starts its beats on notes 0, 2 and 4, so its mask is `0b10101`. A `u64` has room for 64 notes (`MAX_GROUPED_BEATS`), more than any real bar.

Songs and rhythms are `const` tables, so a grouping has to be buildable at compile time too:

```
pub const fn of_groups(groups: &[u8], unit: u8) -> Self {
    let mut mask = 0_u64;
    let mut at = 0;
    let mut index = 0;
    // `for` loops are not allowed in a `const fn` yet: `while` is.
    while index < groups.len() {
        assert!(groups[index] > 0, "a group of no notes");
        assert!(at < Self::MAX_GROUPED_BEATS as usize, "too many notes");
        mask |= 1 << at;
        at += groups[index] as usize;
        index += 1;
    }
    assert!(at > 0 && at <= Self::MAX_GROUPED_BEATS as usize);
    let mut time = TimeSignature::new(at as u8, unit);
    // The usual grouping is written the plain way, so 3+3/8 and 6/8
    // are the same time signature.
    if mask != time.usual_mask() {
        time.groups = mask;
    }
    time
}
```


`1 << at` is a 1 moved `at` places to the left, and `|=` switches that bit on without touching the others. An `assert!` that fails inside a `const fn` evaluated for a constant stops the **build**, not the program, so a mistyped grouping in a song table never reaches anyone. For groupings typed by a person there is `grouped`, which checks the same things and returns an `Option` instead.

The last few lines keep one rule: the usual grouping is stored as 0. So `"3+3/8"` and `"6/8"` give the same value, and `==` says they are equal. Without it the Time menu would show 6/8 as not selected after someone typed 3+3.

The `groups` field is private, so code outside `meter.rs` can only make a grouping through `grouped` or `of_groups`. `beats` and `unit` are public, though, so someone could change `beats` and leave a mask that no longer fits. The private `mask()` covers that: a mask that does not fit the top number is ignored, and the usual beats are used.

Beats of different lengths

In `2+2+3/8` the beats are 4, 4 and 6 steps long, so “which beat is this step in?” can’t be a division. Everything asks `beat_at` instead:

```
pub struct Beat {
    /// Counted from 0: the first beat of the bar is 0.
    pub index: usize,
    /// The step it starts on, counted from the start of the bar.
    pub start: usize,
    /// How many steps it lasts.
    pub length: usize,
}
```

From the test `irregular_meters_group_their_beats`, where `typed` is short for `text.parse().unwrap()`:

```
let balkan = typed("2+2+3/8");
assert_eq!(balkan.beat_starts(), [0, 4, 8]);
assert_eq!(balkan.beat_at(9).length, 6);
assert_eq!(balkan.beat_at(9).index, 2);
```

`beat_at` finds the last beat start at or before the step with `rposition`, which searches a slice from the end.

There is no “steps per beat” function at all, on purpose. Any code that assumed equal beats would be wrong in an irregular meter, and with no such function to call, nobody can make that mistake. (Removing a function is also the quickest way to find every caller: the compiler lists them.)

A bar can be asked about its beats in three ways, because three parts of the program mean different things by “beat”:

Function	Answers	4/4	6/8	7/8	2+2+3/8	5/4
<code>groups</code>	the beats a metronome clicks, in units	1 1 1 1	3 3	1×7	2 2 3	1×5
<code>usual_groups</code>	the unit notes in a usual beat	1	3	1	1	1
<code>felt_groups</code>	the beats a player leans on	2 2	3 3	2 2 3	2 2 3	2 3

`felt_groups` is what the band uses for its “oom-pa”: a grouping if there is one, the three-eighth beats of 6/8, and otherwise beats of two with a three at the end. Bars of three notes or fewer, like 3/4, are one felt beat. How the band plays them is in [The band](#).

Reading and writing “2+2+3/8”

`TimeSignature` implements `Display`, so `format!("{}", time)` writes “3/4”, or “2+2+3/8” with the grouping as the top number, everywhere in the same way. The other direction is the standard `FromStr` trait, which is what makes `"7/8".parse::<TimeSignature>()` work:

```
impl FromStr for TimeSignature {
    type Err = String;

    fn from_str(text: &str) -> Result<Self, Self::Err> {
        let (top, bottom) = text
            .split_once('/')
            .ok_or(tr("write it as two numbers, like 7/8"))?;
        // ... then the bottom number, the top number or numbers, and the
        bar's size
    }
}
```

`split_once('/')` gives `Some((before, after))` or `None`; `ok_or` turns the `None` into an error, and `?` returns it at once. The error is a `String` the Time menu shows as it

is, in red, so the messages go through `tr` and come out in the app's language ([Two languages](#)).

The Time menu's "Any other" row keeps what is being typed in a small `Draft` (the top number, the bottom number, and the grouping as text) in `egui`'s memory between frames, and builds the time signature with the same `parse`. If the groups don't add up to the top number, it says so: "the groups add up to 8, not 11".

Saving it without changing old files

A time signature without a grouping must be saved exactly as before, `(beats: 7, unit: 8)`, so older versions keep opening the file. The attribute `#[serde(from = "SavedTime", into = "SavedTime")]` on `TimeSignature` says: to save one, turn it into a `SavedTime` first; to load one, read a `SavedTime` and turn it back:

```
#[derive(Serialize, Deserialize)]
struct SavedTime {
    beats: u8,
    unit: u8,
    #[serde(default, skip_serializing_if = "Vec::is_empty")]
    groups: Vec<u8>,
}
```

`SavedTime` can hold a friendly `Vec` (`groups: [2, 2, 3]`) because it only lives for a moment while saving or loading. The `From<SavedTime>` conversion leaves out a grouping that does not add up to the top number, so a hand-edited file can't make a bar whose beats don't fit.

Meter: time signatures that change

A song can change time signature partway through: a hymn's last line in 3/4, Mussorgsky's Promenade alternating 5/4 and 6/4. Each change is a marker on the playlist:

```
pub struct TimeMarker {
    /// The step it starts on, counted from the start of the song.
    pub start: usize,
    pub time: TimeSignature,
}
```

A marker always starts a new bar. One placed in the middle of a bar cuts that bar short, which is how a score writes a bar with a beat missing.

With markers, “which bar is step 50 in?” depends on everything before step 50, so every question about bars goes to one type:

```
pub struct Meter {  
    first: TimeSignature,  
    /// Sorted by start, all after step 0, no two on the same step.  
    markers: Vec<TimeMarker>,  
}
```

The comment on `markers` is an *invariant*, a rule that is always true. The fields are private and `Meter::new` makes it true (it drops markers on step 0, sorts the rest and removes duplicates with `dedup_by_key`), so no method has to check it again.

`bar_at` walks the song in **stretches**, one per time signature, counting the bars it passes:

```
4/4 from 0, 3/4 from 32:  
stretch 1: steps 0..32, 16 a bar -> bars 0 and 1  
stretch 2: steps 32.., 12 a bar -> bars 2, 3, 4, ...
```

```

fn stretches(&self) -> impl Iterator<Item = (usize, usize, TimeSignature)>
+ '_ {
    let starts =
        std::iter::once(TimeMarker::new(0,
self.first)).chain(self.markers.iter().copied());
    let ends = self
        .markers
        .iter()
        .map(|marker| marker.start)
        .chain(std::iter::once(usize::MAX));
    starts
        .zip(ends)
        .map(|(from, end)| (from.start, end, from.time))
}

pub fn bar_at(&self, step: usize) -> Bar {
    let mut bars_before = 0;
    for (start, end, time) in self.stretches() {
        let bar = time.steps_per_bar();
        if step < end {
            let index = (step - start) / bar;
            let bar_start = start + index * bar;
            return Bar {
                number: bars_before + index,
                start: bar_start,
                length: bar.min(end - bar_start),
                time,
            };
        }
        bars_before += (end - start).div_ceil(bar);
    }
    unreachable!("the last stretch never ends")
}

```

`stretches` builds an iterator out of others: the starts (the first time signature, then each marker) are `zip` ped with the ends (each marker, then `usize::MAX` for “never”). `div_ceil` divides rounding up, so a short bar before a marker still counts as a bar, and `bar.min(end - bar_start)` gives that short bar its real length. `unreachable!` tells the reader, and the compiler, that the loop always returns: the last stretch ends at `usize::MAX`.

`bars()` gives every bar without end:

```
pub fn bars(&self) -> impl Iterator<Item = Bar> + '_ {
    let mut next = 0;
    std::iter::from_fn(move || {
        let bar = self.bar_at(next);
        next = bar.end();
        Some(bar)
    })
}
```

`std::iter::from_fn` turns a closure into an iterator: each call to `next` runs the closure. It never returns `None`, so the iterator never stops on its own; the playlist draws its grid with `meter.bars().take_while(|bar| bar.start <= steps)` and decides itself how much of it to use.

The rest of `Meter` is the questions the app asks: `time_at`, `bar(number)`, `is_bar_start`, `floor_to_bar`, `floor_to_beat`, `beat_end`, `round_up_to_bar`, and `position(step)` for the BAR:BEAT:STEP display. `Meter::steady(time)` is a song with no markers, and a test checks, step by step, that it answers exactly like the `TimeSignature` alone. `tidy()` drops the markers that change nothing: one that keeps the time signature before it and falls on one of its bar lines.

Who asks the meter

- **The playlist:** its grid, bar numbers, sections and snapping. `Snap::floor` and `Snap::next` give the start and the end of the bar or beat under the pointer, which is what dragging a clip, a section or the loop needs.
- **The channel rack and the piano roll** show one pattern, not the song. They ask `Project::pattern_time(pattern)`: the time signature where the pattern's first clip starts, or, for a pattern not in the song, its own `Pattern::time` (a waltz tried from the Library is in 3/4 even in a 4/4 song), or else the project's.
- **The length of the song**, rounded up to the next bar line.
- **The pattern length menu**, which offers whole bars (`pattern_lengths`: 1, 2, 3, 4, 6, 8, 12 or 16 of them, as long as they fit) and writes "4 bars" instead of "64 steps" (`describe_length`).

The song as it plays

Sections can repeat ([Sections, repeats and loops](#)), so the song as heard is longer than the playlist as drawn. If a 3/4 section plays twice, the song as heard has two 3/4 stretches. The arrangement's *passes* say which part of the timeline plays when, and `Project::played_meter` builds a second `Meter`, in song steps:

```

pub fn played_meter(&self, arrangement: &Arrangement) -> Meter {
    let meter = self.meter();
    let first = arrangement
        .passes
        .first()
        .map_or(self.time_signature, |pass| meter.time_at(pass.from));
    let mut markers = Vec::new();
    for pass in &arrangement.passes {
        markers.push(TimeMarker::new(pass.start,
meter.time_at(pass.from)));
        for marker in meter.markers() {
            if marker.start > pass.from && marker.start < pass.from +
pass.length {
                markers.push(TimeMarker::new(
                    pass.start + marker.start - pass.from,
                    marker.time,
                ));
            }
        }
    }
    Meter::new(first, markers).tidy()
}

```

Every pass gets a marker for the time signature it starts in, plus the markers inside it, and `tidy` then removes the ones that change nothing, such as a 4/4 pass after another 4/4 pass on a bar line. The transport uses this meter for the position display and for Page Up/Page Down, so “the next bar” is the next bar you hear.

The metronome uses it too. `played_clicks` turns it into one `Click` per song step (`Bar`, `Beat` or `None`), and the app sends that list to the audio thread with `Command::SetSongClicks`, so the metronome clicks the beats of each bar: the two dotted quarters of 6/8, the three groups of 2+2+3/8, a higher click on every “one”. In pattern mode it is `pattern_clicks`, in the pattern’s own time. The count-in and the metronome’s sound are in [Practice tools](#).

Tempo changes

Tempo, always valid

```
#[derive(Debug, Clone, Copy, PartialEq, PartialOrd, Serialize,
Deserialize)]
#[serde(from = "f32", into = "f32")]
pub struct Tempo {
    bpm: f32,
}
```

The field is private, so the only way to make a `Tempo` is `Tempo::new`, which clamps the value between `MIN` (10) and `MAX` (522) BPM and turns a NaN into the default tempo, 130. Every `Tempo` in the program is therefore valid. Files go through the same door: a tempo is saved as a plain number, and `From<f32>` reads it back with `Tempo::new`, so a file that says `tempo: 9000.0` opens at 522 ([Saving](#)).

Markers, and a map

A tempo change is a marker on the playlist, like a time signature change:

```
pub struct TempoMarker {
    /// The step it starts on, counted from the start of the song.
    pub start: usize,
    pub tempo: Tempo,
    /// How many steps the change takes; 0 for at once. Project files
    /// leave it out when it is 0.
    #[serde(default, skip_serializing_if = "is_zero")]
    pub glide: usize,
}
```

With `glide` at 0 the tempo changes at once, as after a pause. With a glide it moves a little on every step until it arrives: down for a *ritardando*, up for an *accelerando*. A step is a sixteenth note, so a glide over two bars of 4/4 is 32 small changes, close enough to a curve that nobody hears the stairs. The tempo never changes in the middle of a step.


```
fn bpm_on(&self, step: usize, from: f32) -> f32 {
    let into = step - self.start + 1;
    if into >= self.glide {
        return self.tempo.bpm();
    }
    let done = into as f32 / self.glide as f32;
    from + (self.tempo.bpm() - from) * done
}
```

A glide of 4 steps from 100 to 60 BPM plays 90, 80, 70 and 60: the last step of the glide is already at the new tempo.

`TempoMap` is to tempo what `Meter` is to bars: the first tempo and the markers, with the same invariant, and one main question:

```
pub fn tempo_at(&self, step: usize) -> Tempo {
    // The marker in effect so far, and the tempo its glide started from.
    let mut current: Option<(&TempoMarker, f32)> = None;
    let bpm_on = |current: Option<(&TempoMarker, f32)>, step: usize| match
current {
        Some((marker, from)) => marker.bpm_on(step, from),
        None => self.first.bpm(),
    };
    for marker in self.markers.iter().take_while(|m| m.start <= step) {
        let from = bpm_on(current, marker.start - 1);
        current = Some((marker, from));
    }
    Tempo::new(bpm_on(current, step))
}
```

A glide starts from whatever tempo was playing on the step before its marker, even halfway through another glide, so the loop carries `from` along: each marker asks the one before it “where were you, one step before me?”. `marker.start - 1` can’t underflow, because no marker is on step 0.

The tempo strip

The strip above the playlist’s bar numbers draws the map as a line. It only needs the steps where the line bends (the start, each marker and the end of each glide), joined pairwise with `windows(2)`:

```

let mut bends = vec![0, steps];
for marker in map.markers() {
    bends.push(marker.start);
    bends.push(marker.start + marker.glide);
}
bends.retain(|&step| step <= steps);
bends.sort_unstable();
bends.dedup();
let mut points = Vec::new();
for pair in bends.windows(2) {
    let (from, to) = (pair[0], pair[1]);
    points.push(pos2(layout.x(from), y(map.tempo_at(from))));
    points.push(pos2(layout.x(to),
y(map.tempo_at(to.saturating_sub(1))))));
}

```

Each segment goes from the tempo on its first step to the tempo on its last, so a sudden change becomes a vertical jump and a glide a slope. The labels are written the way a score writes them: `describe` compares the marker's tempo with `tempo_before` its step, and writes “J = 90” for a sudden change, “rit. → 60” for a glide down and “accel. → 140” for a glide up.

The toolbar's TEMPO box sets the tempo the song *starts* at, and `scale_tempo` moves every marker by the same proportion:

```

pub fn scale_tempo(&mut self, tempo: Tempo) {
    let ratio = tempo.bpm() / self.tempo.bpm();
    self.tempo = tempo;
    for marker in &mut self.playlist.tempo_markers {
        marker.tempo = Tempo::new(marker.tempo.bpm() * ratio);
    }
}

```

Practising a piece more slowly keeps its ritardando, only slower too. While a song whose tempo changes is playing, the box shows TEMPO NOW instead: `tempo_at` the playhead's place on the timeline.

A style from the Library brings a tempo of its own, and that one should land where the song really plays, not on its first note: a MIDI file often starts with a few slow notes before its real tempo. So `main_tempo` finds the tempo the song keeps for the most steps, and `take_tempo` scales the whole song so that that one becomes the style's:

```

pub fn take_tempo(&mut self, tempo: Tempo) {
    let ratio = tempo.bpm() / self.main_tempo().bpm();
    self.scale_tempo(Tempo::new(self.tempo.bpm() * ratio));
}

```

`main_tempo` adds up, for each tempo, the steps it holds: from the start to the first marker at the starting tempo, then from each marker to the next. A tempo that comes back later adds to what it held before. Of those held longest, the first wins, so a song with one steady tempo gets that one, and a tie goes to the earlier tempo:

```
let (mut tempo, mut from) = (self.tempo, 0);
for marker in &self.playlist.tempo_markers {
    let start = marker.start.clamp(from, end.max(from));
    hold(tempo, start - from);
    (tempo, from) = (marker.tempo, start);
}
hold(tempo, end.saturating_sub(from));
```

`hold` is a closure that borrows the list `held` mutably, so it is declared `let mut hold:` calling a closure that changes what it captured changes the closure's own state.

`(tempo, from) = (...)` sets two variables at once, a *destructuring assignment*.

Every step at its own tempo

`Project::played_tempos` gives the tempo of every step of the song *as it plays*, so a section that slows down at its end slows down every time round:

```
pub fn played_tempos(&self, arrangement: &Arrangement) -> Vec<Tempo> {
    let map = self.tempo_map();
    if map.is_steady() {
        return Vec::new();
    }
    (0..arrangement.length)
        .map(|step| {
            let timeline =
                arrangement.timeline_step(step).unwrap_or(step);
            map.tempo_at(timeline)
        })
        .collect()
}
```

The list is built on the UI thread and sent with `Command::SetSongTempos`, next to the song. The audio thread may not allocate, and when a new list arrives the old one goes back to the UI thread to be freed, like old songs and old notes ([Two threads](#)). An empty list, the usual case, means “play the TEMPO box’s tempo”, and costs nothing.

A step must last as long as *its own* tempo says, so the tempo has to be set before the step starts. The sequencer’s `due` says which step starts on the next tick ([The sequencer](#)), and the audio thread asks it first, on every frame:

```

if let Some(step) = self.sequencer.due() {
    // ...
    let tempo = self.tempo_of_step(step);
    if tempo != self.tempo_now {
        self.set_tempo(tempo);
    }
}
if let Some(step) = self.sequencer.tick() {

```

`tempo_of_step` is the song's tempo for that step in song mode, and the TEMPO box's in pattern mode, where you work on one pattern and want a steady loop. (The part left out above, `lap_tempos`, is the speed trainer's next tempo, which takes over when the loop comes round: [Practice tools](#).)

Notes already sounding have a length in frames, worked out at the old tempo.

`set_tempo` stretches them so they still end on the step they should:

```

fn set_tempo(&mut self, tempo: Tempo) {
    self.tempo_now = tempo;
    let before = self.sequencer.step_frames();
    let next_step = self.sequencer.frames_to_next_step();
    self.sequencer.set_tempo(tempo);
    let stretch = self.sequencer.step_frames() / before;
    for channel in &mut self.channels {
        channel.instrument.tempo_changed(next_step, stretch);
        channel.effects.set_tempo(tempo);
    }
    // ...
}

```

and each instrument works out what is left of every held note:

```

pub(crate) fn stretched(left: f64, next_step: f64, stretch: f64) -> f64 {
    if left <= next_step {
        left
    } else {
        (next_step + (left - next_step) * stretch).round()
    }
}

```

The frames up to the next step belong to the step already playing, which keeps its length; only the rest is stretched. A note with 10 000 frames left, 2 000 of them in this step, at half the tempo (a stretch of 2) has $2\,000 + 8\,000 \times 2 = 18\,000$ left. Effects that follow the beat, such as a delay set in note values, get the new tempo too ([The mixer](#)).

An export plays the song through the same audio thread, so it gets the same list. Its length is no longer “steps times the length of a step”: `played_seconds` adds up every

step at its own tempo, with `tempo::seconds`:

```
pub fn seconds(steps: usize, tempo: impl Fn(usize) -> Tempo) -> f64 {
    (0..steps)
        .map(|step| f64::from(tempo(step).seconds_per_beat()) / 4.0)
        .sum()
}
```

`impl Fn(usize) -> Tempo` takes any closure from a step to a tempo, so the same function serves a `TempoMap` in a test and the played list in `played_seconds`.

Putting a song in another time signature

A time signature only moves bar lines. But pick a waltz in the Library while a 4/4 song plays, and its twelve-step bars laid into sixteen-step ones play “one two three” and then a beat of silence, every bar. To sound like a waltz, every bar of the song has to become a 3/4 bar. `remeter` does that, with one rule: **bar n of the old song becomes bar n of the new one, and every note keeps its place in its bar.**

```
4/4: | C . . . E . . . G . . . E . . . | 16 steps
3/4: | C . . E . . G . . E . . | 12 steps
```

It is a mechanical stretch, the way a tape machine would do it, not the way a composer would rewrite a tune for a new meter.

Squeezing a bar, to the tick

Squeezing sixteen steps into twelve puts many notes between steps: eighth notes two steps apart become 1.5 steps apart. Rounding them to the grid would make them uneven. So the squeeze is done in **ticks**, 24 to a step (`TICKS_PER_STEP` , the unit MIDI import uses for notes off the grid, see [MIDI](#)):

```

pub struct Remap {
    old: Meter,
    /// Steps in one new bar.
    bar: usize,
}

impl Remap {
    pub fn tick(&self, tick: usize) -> usize {
        let bar = self.old.bar_at(tick / TICKS_PER_STEP);
        let start = bar.start * TICKS_PER_STEP;
        let length = bar.length * TICKS_PER_STEP;
        let new_bar = self.bar * TICKS_PER_STEP;
        // Rounded to the nearest tick.
        bar.number * new_bar + ((tick - start) * new_bar + length / 2) /
length
    }
}

```

Two habits of integer arithmetic are in the last line. **Multiply before dividing:** `(tick - start) / length * new_bar` would be 0 for every tick inside the bar. **Add half before dividing** to round to the nearest whole number instead of always down. The second eighth of a 4/4 bar, tick 48 of 384, lands on $(48 \times 288 + 192) / 384 = 36$ of a 288-tick 3/4 bar: 1.5 steps, exactly.

A `Remap` holds the song's old `Meter`, so it works for songs that already change meter: every old bar, whatever its time signature (a short one too), becomes one whole new bar. `Remap::step` is the same, rounded to the nearest step, for things that live on steps.

Every bar keeps its duration

The tempo counts quarter notes. A 3/4 bar has three, a 4/4 bar four, so at the same tempo the 3/4 song would be over in three quarters of the time. So `remeter` scales the tempo by the same ratio as the bar: 4/4 at 112 becomes 3/4 at 84, and a bar lasts 2.14 seconds in both. Tempo markers are scaled the same way, and a meter change in the middle of the song that changed the bar length gets a tempo marker of its own, so each stretch keeps its pace.

The scaling has to come back exactly. 108.5 BPM in 7/8, put back into 4/4, must be $108.5 \times 16 / 14 = 124$, not 124.00001. An `f32` has about seven significant digits, and the ratio 16/14 has no exact value in it. So the code multiplies first and divides last, in `f64`:

```
let scaled = |tempo: Tempo, step: usize| {
    Tempo::new((f64::from(tempo.bpm()) * bar as f64 / old_bar(step) as
f64) as f32)
};
```

$108.5 \times 16 = 1736$ is exact, and $1736 / 14 = 124$ is exact too. A test takes several tempos through 7/8, 3/4, 13/16, 6/8 and 12/8 and back to 4/4, and checks they come back unchanged.

So after `remeter` every note still starts at the same moment, in seconds, and lasts as long. The music sounds the same; only how it is written and counted changes. What makes a song *sound* like a waltz is the drums and the way the chords are struck, which is why the Library and the Time box re-meter the song **and** give it a rhythm in the new meter. That choice of rhythm is in [The Library](#).

Patterns that grow too long

A pattern holds at most 256 steps (`MAX_STEPS`). Sixteen bars of 4/4 are exactly that, but sixteen bars of 12/8 are 384. Rather than drop notes, `remeter` cuts such a pattern into pieces of whole bars that fit:

```
let chunk = MAX_STEPS / bar * bar;
```

$256 / 24 \times 24$ is 240 steps: ten bars of 12/8, then six. The pieces are named “Theme”, “Theme (2)” and so on, and each clip of the pattern becomes clips of the pieces, one after the other. The song sounds the same; the playlist shows one clip more.

Everything else moves with its bar

Clips, sections, the loop, key markers and the band’s markers all go through `song.step`. A tempo marker’s glide is mapped by its two ends, so a ritardando over two bars of 4/4 lasts two bars of 3/4. Chords change on a beat, so they go to the nearest beat of the new bar (`snap_to_beat`). The time markers go: the whole song is in one time signature now.

Two markers one step apart can be squeezed onto the same step. `Vec::dedup_by` removes neighbours its closure calls equal; the closure gets the later element first and the earlier second, and may change the earlier one, which is how “the later marker wins” is written:

```
playlist_mut.tempo_markers.dedup_by(|later, earlier| {
    later.start == earlier.start && {
        *earlier = *later;
        true
    }
});
```

`a && { ... }` runs the block only when `a` is true: the copy happens only for markers on the same step.

`remeter_song` also works out every new pattern from the untouched project first, keeps a clone of the old playlist, and only then writes. Reading the project while writing it would need two borrows of the same `Project` at once, one shared and one mutable, which the borrow checker rightly refuses. And it runs inside `project.moving_the_band(...)`: the band's notes are moved along with everything else, and that is not an edit of yours, so the band keeps following the chord sheet ([The band](#)).

Keeping your place

While the song plays you are somewhere, bar 13 beat 2, say. After the squeeze, step 196 is no longer in bar 13. So after any change of meter the app finds the bar the playhead was in, and how far into it, and moves to the same bar and the same fraction of it:

```
let place = |step: usize| {
    let bar = old.bar_at(step);
    let into = (step - bar.start) as f64 / bar.length.max(1) as f64;
    let target = new.bar(bar.number);
    target.start + (into * target.length as f64).round() as usize
};
```

`old` and `new` are the `played_meter`s before and after, so repeats count. The same closure moves where Play starts and where Stop returns to. It works after undo and redo too, which can swap in a song with time markers of its own, where a `Remap` would not fit.

Swing

Swing plays the two eighth notes of each beat long-short. The project has one amount, from 0 ("straight") to `MAX_SWING`, 100 (a triplet's two thirds and one third):


```
pub swing: u8,
```

The notes in the patterns never move. Instead, everything that *plays* the song asks for the notes as they are played:

```
pub fn played_notes(&self, pattern: usize, channel: usize) -> Cow<'_,
[NoteEvent]> {
    let Some(written) = self.pattern(pattern).map(|p| p.notes(channel))
else {
    return Cow::Borrowed(&[]);
};
match self.pattern_swing(pattern) {
    0 => Cow::Borrowed(written),
    swing => Cow::Owned(swung(written, swing,
self.plays_hits(channel))),
}
}
```

`Cow` ("clone on write", from `std::borrow`) is either **borrowed** or **owned**. Most songs don't swing, and for them `played_notes` hands back the pattern's own notes without copying one. Only a swinging pattern gets a new `Vec`. Both are read the same way, as a slice, and a caller that needs a `Vec` of its own calls `.into_owned()`, which copies only if it was borrowed. The `'_` says the borrowed case borrows from `self`: the result can't outlive the project.

The audio thread, the MIDI export and the accompaniment that follows your playing take `played_notes`. The piano roll, the step buttons and the chord sheet read the written notes, so the grid you edit stays a grid.

Moving the second eighth

A beat is 96 ticks. The second eighth starts on tick 48; the third note of a triplet on tick 64. So full swing is 16 ticks late:

```
const BEAT: usize = 4 * TICKS_PER_STEP;
const OFFBEAT: usize = BEAT / 2;
const FULL_DELAY: usize = BEAT * 2 / 3 - OFFBEAT;
const NEAR: usize = BEAT / 16;

pub fn delay(swing: u8) -> usize {
    let swing = usize::from(swing.min(MAX_SWING));
    (FULL_DELAY * swing + usize::from(MAX_SWING) / 2) /
    usize::from(MAX_SWING)
}
```

Adding half of 100 before dividing by 100 rounds to the nearest tick again: 50 is 8 ticks, a 7:5 swing; 70 is 11.2 and becomes 11.

`swung` moves every start and end that falls on or near an offbeat:

```
pub fn swung(notes: &[NoteEvent], swing: u8, hits: bool) -> Vec<NoteEvent>
{
    let late = delay(swing);
    let played = |tick: usize| match tick % BEAT {
        offset if offset.abs_diff(OFFBEAT) < NEAR => tick + late,
        _ => tick,
    };
    notes
        .iter()
        .map(|note| {
            let start = played(note.start_tick());
            let end = match hits {
                true => note.end_tick(),
                false => played(note.end_tick()),
            };
            NoteEvent {
                velocity: note.velocity,
                ..NoteEvent::exact(note.pitch, start,
end.saturating_sub(start))
            }
        })
        .collect()
}
```

The first arm is a **match guard**: `offset` is a new variable that matches anything, and the `if` after it decides whether the arm applies. “Near” is less than six ticks either way, because the strings of a strummed chord start a tick apart, 48, 49, 50, and all of them must move together. `abs_diff` is the distance between two numbers whichever is bigger; `offset - OFFBEAT` on a `usize` below 48 would underflow and panic in a debug build.

A pattern of a constant is different from a binding. Had the arm been `OFFBEAT => ...`, it would compare with the constant’s value, 48. Had it been a lowercase `offbeat => ...`, Rust would read it as a new variable matching *everything*, and the `_` arm would never run. The compiler warns about that, and the convention of capitals for constants keeps the two apart.

A note that ends on the offbeat ends later too, so the first eighth really is long, legato into the second. Drum hits are different: a sampled drum sounds to its end however long the note is, so only their starts move (`hits`). The new note is built with `NoteEvent::exact`, the same function MIDI import uses for notes off the grid: a swung eighth on step 2 becomes step 2 with a `nudge` of 16 ticks, which the audio

thread already plays late ([The sequencer](#)). Notes that are not on an offbeat (sixteenths, a dotted eighth and a sixteenth) stay as they are.

Where swing applies

```
pub fn swings_in(time: TimeSignature) -> bool {  
    time.unit <= 4  
}
```

Only time signatures counted in quarter notes or longer have pairs of eighths in a beat. In 6/8 or 12/8 a beat is already three eighths, and in 5/8 or 7/8 the eighth is the beat. `pattern_swing` returns 0 for a pattern in such a meter, and the Swing control is greyed out when no part of the song can swing, with a hint that says why. A band section played in another style can bring that style's swing (`section_swing`).

No new command for the audio thread

The audio thread learns about every edit from `diff`, which compares the project before and after, and it compares the *played* notes ([Telling the audio thread what changed](#)). So changing the swing needs no command of its own: the patterns whose played notes moved are sent again, and the others are left alone. A pattern moved into 6/8, which stops swinging, is caught the same way, and so is undo, which swaps a whole earlier project in. When nothing swings, both sides of the comparison are borrowed `Cow`s, and comparing them compares the slices without copying anything.

Remembered per style

Each rhythm in the Library carries its own swing (`Rhythm::swing`), and a style takes its rhythm's: Swing, Jazz ballad, Jazz waltz, Shuffle and Five-four swing at 100, Rock and roll at 50, the rest straight. Their drum patterns are written in even eighths, as sheet music writes them, and the swing does the rest. When you change the amount, the app remembers it for that style, between runs, in `app/swing.rs`:

```
#[derive(Debug, Default)]
pub(super) struct SwingMemory {
    amounts: BTreeMap<String, u8>,
    /// Whether an amount changed since they were last stored.
    changed: bool,
}

/// What is stored: a newtype, so it has a type of its own in egui's
/// store.
#[derive(Debug, Clone, Default, Serialize, Deserialize)]
#[serde(default)]
struct Remembered(BTreeMap<String, u8>);
```

egui keeps persisted values by an `Id` and a type. Wrapping the map in a struct of its own, a **newtype**, gives it a type nobody else uses, so a `BTreeMap<String, u8>` stored by another part of the app could never be read as this one. A `BTreeMap` keeps its keys sorted, so the settings file lists the styles in the same order every time. `store_swings` runs once a frame and writes only when `changed` is set.

Picking a style first gives the song the style's own swing, in `lineup-core`, which knows nothing about settings; then the app puts the remembered amount in its place, in the same undo step.

Loose timing

A good player is never exactly on the grid. **Loose**, in the playlist's toolbar, sets how far off it notes may land, from *off* to 40 ms (`MAX_LOOSE_TIMING`). It is saved in the project as `loose_timing`, in milliseconds, and the audio thread turns it into frames once, when `Command::SetLooseTiming` arrives.

Each note's delay is worked out when its step starts:

```
fn delay(loose_frames: f64, channel: usize, song_step: usize, pitch: Note)
-> f64 {
    let onset = ((song_step as u64) << 8) | channel as u64;
    let together = scatter(onset);
    let apart = scatter(onset ^ (u64::from(pitch.midi()) << 40));
    loose_frames * (0.8 * together + 0.2 * apart)
}
```

A pianist is late or early by about the same amount for all the notes of a chord, so four fifths of the delay are shared by every note a channel starts on a step, and one fifth differs from note to note. The numbers come from `scatter`, which mixes the bits of a seed with the `SplitMix64` function:

```
fn scatter(seed: u64) -> f64 {
    let mut x = seed.wrapping_add(0x9E37_79B9_7F4A_7C15);
    x = (x ^ (x >> 30)).wrapping_mul(0xBF58_476D_1CE4_E5B9);
    x = (x ^ (x >> 27)).wrapping_mul(0x94D0_49BB_1331_11EB);
    x ^= x >> 31;
    // The top 53 bits, as many as an `f64` holds exactly.
    (x >> 11) as f64 / (1u64 << 53) as f64
}
```

It looks random, but the same seed always gives the same number, so a song plays the same way every time and a WAV export sounds like playback. `wrapping_mul` lets the multiplication wrap past `u64::MAX` instead of panicking in a debug build: here wrapping is the point. `scatter` also needs no state, which suits the audio thread: no random generator to keep, lock or seed.

A delayed note waits in a list until its frame:

```
pub(crate) struct Pending {
    channel: usize,
    origin: Origin,
    pitch: Note,
    velocity: u8,
    gate: u32,
    /// The frame it starts on, counted on [AudioThread::now`]'s clock.
    at: u64,
    /// How many frames after its step that is, for a trace.
    late: u32,
}
```

The list is made with room for 512 notes (`PENDING_NOTES`) on the UI thread, and the audio thread never lets it grow:

```
// A full list never grows (that would allocate): the note
// then starts on time.
if late > 0 && pending.len() < pending.capacity() {
```

Every frame, the notes whose `at` has come are started, and taken out with `swap_remove`, which moves the last note into the gap instead of shifting the whole list down. The order of waiting notes does not matter, so the cheaper removal is the right one.

Why only late, never early? To start a note early the audio thread would have to look into the next step before it starts. Late by 0 to 20 ms sounds the same as early or late by 10 ms around a grid that is itself 10 ms late, and nobody hears a whole song 10 ms late.

In files and MIDI files

Each of these features is saved only when it is used, so a project that doesn't use it can still be opened by older versions ([Saving](#)). `format_for` writes the oldest format that can hold the project:

Format	Needed for
3	A time signature other than 4/4, or a pattern with its own
5	Time markers
7	Tempo markers, or loose timing
10	Notes off the grid, such as a re-metered song's
11	A grouped time signature, such as 2+2+3/8
16	Swing

When a file is read, every time and tempo marker goes through `set_time_marker` or `set_tempo_marker`, which refuse one that starts too late or whose bars would not fit in a pattern, as if it had been added by hand.

A Standard MIDI File keeps tempo and meter as *meta events* on its first track. A tempo is written as microseconds per beat, in three bytes, and a change gets an event on the step where the played tempo changes, so a glide becomes one event per step, which is how MIDI files write a ritardando:

```
let tempo_event = |tempo: Tempo| {
  let micros = (60_000_000.0 / f64::from(tempo.bpm())).round() as u32;
  micros.min(0xFF_FFFF).to_be_bytes()[1..].to_vec()
};
// ...
for (step, pair) in tempos.windows(2).enumerate() {
  if pair[1] != pair[0] {
    let tick = (step as u32 + 1) * TICKS_PER_STEP;
    events.push((tick, TEMPO, tempo_event(pair[1])));
  }
}
```

`to_be_bytes` gives the four bytes of the `u32`, most significant first (“big-endian”, the order MIDI files use), and `[1..]` drops the first, which is always 0 below `0xFF_FFFF`.

A time signature event holds the top number, the bottom number as a power of two (`unit.ilog2()`: 2 for a quarter, 3 for an eighth), how many MIDI clocks a metronome click lasts (24 to a quarter note, so 6 per step), and a constant 8. There is one at the start and one at every change of the played meter. MIDI has no room for a grouping,

so a grouped bar clicks every note (`ungrouped().first_beat()`). How a whole MIDI file is written is in [MIDI](#).

Testing it

Each type has tests next to it that read like the rules they check:

- `meter.rs`: `a_steady_meter_counts_like_its_time_signature`,
`a_marker_off_the_bar_line_leaves_a_short_bar`,
`the_usual_grouping_is_written_plainly` (3+3/8 is 6/8),
`a_grouping_is_saved_only_when_there_is_one`,
`nonsense_does_not_divide_by_zero`.
- `tempo.rs`: `a_glide_moves_a_little_every_step` (the 90, 80, 70, 60 above),
`a_glide_starts_from_where_the_last_one_got_to` (a *ritardando* cut short by an *a tempo*), `seconds_add_up_every_step`.
- `remeter.rs`: `every_note_keeps_its_place_in_the_bar`,
`notes_between_steps_keep_their_exact_place`,
`every_bar_last_as_long_as_before` (the song's length in seconds, before and after, within 0.2%),
`a_tempo_put_in_another_meter_and_back_is_the_same_tempo`,
`a_pattern_too_long_for_the_new_bars_is_cut`.
- `swing.rs`: `full_swing_plays_the_eighths_as_triplets`,
`half_swing_is_seven_to_five`, `a_strum_on_the_offbeat_moves_as_a_whole`,
`notes_off_the_eighths_stay`.
- `engine.rs` plays a few notes and checks where the sound starts, frame by frame: `a_song_plays_each_step_at_its_own_tempo` (two steps at 120 BPM, then two at 60, which last twice as long) and
`loose_timing_starts_notes_a_little_late` (every note within 960 frames, 20 ms at 48 kHz, after its step; not all by the same amount; and the same every time).

One test goes through the whole app:

`styles_tried_one_after_another_keep_the_songs_tempo` opens El Choclo and takes it through seven styles and meters (among them a tango vals in 3/4 and the Time box at 7/8), in pattern mode, in song mode stopped, and in song mode playing. After each one, the tempo must be unchanged when the meter is, and the song's own whenever it is back in 4/4. Then it plays the start of the song with the [playback trace](#) and checks that every melody note starts and lasts as it did, within 2 ms. To keep it quick in a debug build, it plays only twelve seconds, at 8000 samples a second: plenty to time notes.

Instruments: drums, sampler and Lineup Osc

Files: `crates/lineup-audio/src/instrument.rs`, `crates/lineup-audio/src/drums.rs`, `crates/lineup-audio/src/dsp.rs`, `crates/lineup-audio/src/sample.rs` and `crates/lineup-audio/src/timbre.rs`

Every channel has an instrument: the part that turns notes into sound. This chapter is about the two that Ferrous Lineup makes itself, the **sampler**, which plays a recorded or synthesized sound, and the **Lineup Osc**, a small synthesizer. The other two kinds play through something else: a SoundFont ([SoundFonts and General MIDI](#)) or an external synth over MIDI ([MIDI](#)). How the sounds are chosen in the app is in the manual's [Sounds](#).

One enum, four instruments

```
pub enum Instrument {
    Sampler(Sampler),
    Synth(Synth),
    SoundFont(SoundFontPlayer),
    MidiOut(MidiOut),
}
```

A common design for “many kinds of one thing” is a trait, `trait Instrument`, with values of type `Box<dyn Instrument>`. That is open: anyone could add a kind without touching this file. But every `Box::new` allocates, and every call goes through a pointer to a table of functions. With four kinds, all known, an enum is simpler. The audio thread `match` es on it, making one never allocates, and when a fifth kind is added the compiler lists every `match` that has to learn about it.

Every method of `Instrument` is a `match` that hands the call to the kind inside:

```
pub fn release_all(&mut self) {
    match self {
        Instrument::Sampler(sampler) => sampler.release_all(),
        Instrument::Synth(synth) => synth.release_all(),
        Instrument::SoundFont(player) => player.release_all(),
        Instrument::MidiOut(midi_out) => midi_out.release_all(),
    }
}
```


What an instrument can be asked:

Method	What it does
<code>note_on</code>	Starts a note: a pitch, a velocity and a gate , how many frames it is held
<code>note_on_from</code>	The same, for a note of the song, which remembers where it came from (<code>origin</code> , see The sequencer)
<code>note_off</code>	Lets go of a note held with the gate <code>HELD</code>
<code>release_all</code>	Lets go of every held note, when playback stops
<code>release_unless</code>	Lets go of the song's notes that the song no longer plays
<code>resume</code>	Starts a long sound partway through, where it would be had playback started earlier (the sampler only)
<code>tempo_changed</code>	Stretches held notes after a change of tempo (Tempo , meter , swing and feel)
<code>next_sample</code>	The next frame of sound: the sum of every voice

A note of the song knows its length when it starts, so it gets a gate in frames and ends by itself; the audio thread never has to remember to stop it. A key pressed on a MIDI keyboard has no known length, so it gets `HELD` , `u32::MAX` , which never runs out: only `note_off` ends it.

The instrument for a channel is built by `Instrument::for_channel` , on the UI thread, from the project's `Channel` :

```
Ok(match &channel.kind {
  // A drum sent over MIDI is its note of a General MIDI drum set.
  ChannelKind::Sampler(SampleSource::Builtin(drum)) => match
channel.sent_on {
  Some(number) => Instrument::MidiOut(

MidiOut::new(number).playing_as(Some(general_midi_drum(*drum))),
  ),
  None => {
    Instrument::Sampler(Sampler::new(Arc::new(drums::render(*drum,
sample_rate))))
  }
},
ChannelKind::Sampler(SampleSource::File(path)) => {
  Instrument::Sampler(Sampler::new(samples.get(path)?))
}
ChannelKind::Synth => Instrument::Synth(Synth::with_timbre(
  sample_rate,
  Timbre::for_program(channel.program),
)),
ChannelKind::MidiOut { channel: number } => {

Instrument::MidiOut(MidiOut::new(*number).with_program(channel.program))
}
})
```

Drums are synthesized here and files read here (through the [SampleCache](#)), which is slow work that may allocate: exactly what the audio thread must not do. The finished instrument is sent over in a command.

Drums from a few lines of math

Ferrous Lineup ships no sound files. Its thirteen drums are computed when a drum channel is created, each in a few lines:

Drum	Recipe
Kick	A sine whose pitch falls fast from 150 to 45 Hz, fading over half a second
Snare	A short 185 Hz tone (the shell) plus high-passed noise (the wires)
Clap	High-passed noise in three bursts 11 ms apart, like several hands, and a tail
Hi-hat	Very bright noise, high-passed at 7 kHz, gone in about a tenth of a second

Drum	Recipe
Open hat, ride, crash	"Metal" (below) and noise, ringing longer and longer
Rim	A 1750 Hz click, a 480 Hz body and a snap of noise
Low and high tom	Like the kick, higher and rounder, settling at 98 and 165 Hz
Cowbell	Two square waves at 540 and 800 Hz, a little less than a fifth apart
Shaker	Bright noise that swells for 15 ms, then dies away quickly
Conga	A 320 Hz tone that drops a little, and the slap of a hand

Most of them share one shape, the **exponential decay**:

```
fn decay(t: f32, tau: f32) -> f32 {
    (-t / tau).exp()
}
```

It is 1 at the start and falls to about 37% after `tau` seconds, 13% after twice that, and so on. Struck things fade this way.

Closures that remember: FnMut

All the drums are built with one helper:

```
fn generate(seconds: f32, sample_rate: u32, f: impl FnMut(f32) -> f32) ->
Vec<f32> {
    let frames = (seconds * sample_rate as f32) as usize;
    (0..frames)
        .map(|i| i as f32 / sample_rate as f32)
        .map(f)
        .collect()
}
```

It takes a closure `f` from a time in seconds to a sample. The kick's closure keeps a running `phase` between calls:

```
fn kick(sample_rate: u32) -> Vec<f32> {
    let mut phase = 0.0;
    generate(0.5, sample_rate, |t| {
        let frequency = 45.0 + 105.0 * decay(t, 0.04);
        phase += frequency / sample_rate as f32;
        (phase * TAU).sin() * decay(t, 0.15)
    })
}
```

The closure *changes* `phase`, a variable it captured from `kick`. A closure that changes what it captures implements the `FnMut` trait; `Fn` is for closures that only read, and `FnOnce` for those that can be called only once. `impl FnMut(f32) -> f32` in the parameter list means “any closure of that shape”, and the compiler makes a copy of `generate` for each drum’s closure, as it does for generic types.

Why add up a phase instead of computing `sin(t * frequency * 2π)`? Because the frequency changes over time. The phase is the sum of all the small steps taken so far, so the pitch sweeps smoothly; the direct formula would jump around as the frequency changed.

Metal from six square waves

The cymbals use a trick from the analog drum machines of the early 1980s: six square waves at frequencies that are not multiples of each other, added together.

```
const METAL: [f32; 6] = [205.3, 304.4, 369.6, 522.7, 540.0, 800.0];

fn metal(t: f32, ratio: f32) -> f32 {
    METAL.iter().map(|&f| square(t, f * ratio)).sum::<f32>() / METAL.len()
as f32
}
```

Notes whose frequencies are simple multiples of each other blend into a chord. These don’t, so together they clash and shimmer like metal. A high-pass filter keeps the bright part, and the envelope decides whether it is a closed hat, an open hat, a ride or a crash.

Every drum is then scaled with `normalize` so that its loudest sample is 0.9, and two tests go through `Drum::ALL`: every drum reaches that peak and is longer than a thousand frames (`every_drum_is_audible_and_in_range`), and every drum ends quietly instead of being cut off (`every_drum_fades_out`).

Noise and filters

`dsp.rs` holds the small building blocks. **White noise** comes from xorshift, a tiny random number generator:

```
pub fn next_sample(&mut self) -> f32 {
    let mut x = self.state;
    x ^= x << 13;
    x ^= x >> 17;
    x ^= x << 5;
    self.state = x;
    (x as f32 / u32::MAX as f32) * 2.0 - 1.0
}
```

Three shifts and three exclusive-ors. It is no good for security, but it sounds just like noise, never allocates, needs no crate, and gives the same “random” drums every time, which makes them testable. Its one trap is zero: xorshift from 0 stays at 0 for ever, so `Noise::new` starts from `seed.max(1)`.

The filters are **one-pole** filters, the simplest there are. The low-pass moves its output a fraction of the way toward the input on every sample, so fast changes (high frequencies) are smoothed away:

```
pub fn process(&mut self, input: f32) -> f32 {
    self.state += (input - self.state) * self.coefficient;
    self.state
}
```

That is the same arithmetic as `Smoothed`, the volume that glides instead of jumping ([Two threads](#)). The high-pass follows the *changes* in its input instead, so a steady signal dies away and fast changes pass. The tests say exactly that: a constant input goes through the low-pass and is removed by the high-pass.

The sampler

```
pub struct Sampler {
    sample: Arc<SampleData>,
    /// Boxed so an `Instrument` stays small: commands carrying one are
    /// copied into the command queue, and a pointer is cheaper to copy
    /// than eight voices. The box is created here, on the UI thread.
    voices: Box<[SamplerVoice; SAMPLER_VOICES]>,
    notes_started: u64,
}
```

`Arc<T>` (“atomically reference-counted”) is a pointer that several owners share. Cloning one does not copy the data, it adds one to a counter, and the data is freed when the last `Arc` goes. The counter is updated atomically, so it is safe across threads, which is what lets a sample be built on the UI thread and played on the audio thread. The same sound file in three channels is in memory once.

Each of the eight voices plays one hit:

```
fn next_sample(&mut self, frames: &[f32]) -> f32 {
    let Some(position) = self.position else {
        return 0.0;
    };
    let index = position as usize;
    if index + 1 >= frames.len() {
        self.position = None;
        return frames.get(index).copied().unwrap_or(0.0) * self.gain;
    }
    // Linear interpolation between the two nearest frames.
    let fraction = (position - index as f64) as f32;
    let sample = frames[index] + (frames[index + 1] - frames[index]) *
fraction;
    self.position = Some(position + self.increment);
    // ...
}
```

`let ... else` matches a pattern, or runs the `else` block, which must leave the function. It keeps the main path unindented: “not playing, so silence; otherwise carry on with `position`”.

`position` is a fraction of a frame, because a sound rarely moves through its frames at exactly one per output frame. A WAV recorded at 44.1 kHz played on a 48 kHz sound card must move a little slower, or it would sound higher and shorter. And a sampler plays pitches by playing faster or slower: an octave up is twice as fast, and each semitone multiplies the speed by the twelfth root of two, the same rule

`Note::frequency` uses ([Pitch classes, notes and spelling](#)):

```
// Every semitone up plays the sample 2^(1/12) times faster.
let semitones = pitch.midi() as f64 - STEP_NOTE.midi() as f64;
let pitch_ratio = 2.0_f64.powf(semitones / 12.0);
let rate_ratio = self.sample.sample_rate() as f64 / output_rate as f64;
```

The voice’s `increment` is the two ratios multiplied. On `STEP_NOTE`, C4, the note a step button plays, the pitch ratio is 1 and the sound plays as recorded. Between two stored frames the voice draws a straight line, **linear interpolation**: not the best a sampler can do, but cheap and good enough for drums.

The velocity scales the volume, `velocity / 127`. A sampler ignores the gate: like a drum machine, it plays its whole sound every time, and because it has voices, a new hit does not cut off the one before.

That is right for drums and wrong for one thing: a long recording, such as a whole song imported as audio, would play on after Stop. So `release_all` fades out the voices of a sound longer than ten seconds, in about 15 ms, and leaves short hits ringing:

```
pub fn release_all(&mut self) {
    let long = self.sample.frames().len() as u64
        > u64::from(self.sample.sample_rate()) * LONG_SAMPLE_SECONDS;
    if long {
        for voice in self.voices.iter_mut().filter(|voice|
!voice.is_idle()) {
            voice.fading.get_or_insert(1.0);
        }
    }
}
```

`release_unless`, for a clip taken away or a track muted while it plays, fades the song's sounds whatever their length.

Starting partway

A note sounds when playback reaches the step it starts on. For a drum hit that is all there is to it, but a recording is one note on the first step of a song: started from bar 20, the song would play without it. So when playback starts, jumps, or comes round to the start of the loop, the engine's `resume_long_sounds` looks back over the song for the notes that started before the playhead, and asks each instrument to `resume` them:

```

fn resume(
    &mut self,
    pitch: Note,
    velocity: u8,
    output_rate: u32,
    origin: Origin,
    elapsed: f64,
) -> bool {
    let increment = self.increment(pitch, output_rate);
    let position = elapsed * increment;
    if !self.is_long() || position + 1.0 >= self.sample.frames().len() as
f64 {
        return false;
    }
    self.start_at(position, increment, velocity, Some(origin));
    true
}

```

`elapsed` is how many output frames ago the note started, and a voice moves `increment` frames of the sound for each output frame, so the voice starts `elapsed * increment` frames in, as far as it would have got. Only a long sound, over ten seconds, is started this way, and only if it lasts that long: a drum hit from two bars ago has long finished, and other instruments answer `false`. `resume_long_sounds` goes through every piece of a clip that starts at or before the step, and every time the piece played each note:


```

for note in channel.notes.get(play.pattern) {
  // Every time the piece played the note before
  // `step`, counted from the piece's start, which is
  // below zero for a time in the part of the clip the
  // piece skips.
  let first = play.start as isize - play.skip as isize + note.start as
  isize;
  let times = (first..play.end().min(step) as isize).step_by(length);
  for at in times.filter(|&at| at < step as isize) {
    let origin = Origin {
      pattern: play.pattern,
      pattern_step: note.start,
      song_step: at.max(0) as usize,
      over_song: false,
    };
    let elapsed = frames_between(timing, at, step);
    let started = channel.instrument.resume(
      origin,
      note.pitch,
      note.velocity,
      self.sample_rate,
      elapsed,
    );
    // ...
  }
}

```

`first` is where the note falls if the piece started at its own first step; it is below zero when the clip leaves out its start (`skip`), and `step_by(length)` gives each repeat of the pattern within the piece. `frames_between` adds up the frames of the steps from there to the playhead at the song's tempo on each step, so a recording under a tempo change starts at the right place too.

Loading a WAV file

`SampleData::load_wav` reads WAV files with the `hound` crate. A WAV stores samples as floats or as integers of 8 to 32 bits, and both become floats between -1 and 1:

```

let interleaved: Vec<f32> = match spec.sample_format {
  hound::SampleFormat::Float => reader
    .samples::<f32>()
    .collect::<Result<_, _>>()
    .map_err(SampleError::Wav)?,
  hound::SampleFormat::Int => {
    let full_scale = (1_i64 << (spec.bits_per_sample - 1)) as f32;
    reader
      .samples::<i32>()
      .map(|sample| sample.map(|s| s as f32 / full_scale))
      .collect::<Result<_, _>>()
      .map_err(SampleError::Wav)?
  }
};

```

`samples()` gives a `Result` per sample, because a file can be cut off halfway. Collecting an iterator of `Result`s into a `Result<Vec<_, _>, _>` gives `Ok` with every value if all were `Ok`, or the first `Err`, and stops there. The `_`s let the compiler fill in the types. Then `interleaved.chunks(channels)` takes one frame at a time (left, right) and averages it, so a stereo file becomes mono.

`SampleError` implements `Display` with translated messages, so “could not read the WAV file” reaches the user in their language. The test `loads_and_mixes_to_mono` writes a real 16-bit stereo WAV to the system’s temporary folder with `hound::WavWriter` and reads it back.

Voices

A **voice** is one note’s worth of sound-making: its own place in the sound, its own oscillator, its own envelope. The Lineup Osc has 16 voices and a sampler 8. Each new note takes one:

```

trait Voice {
    /// Whether the voice is silent and free to play a new note.
    fn is_idle(&self) -> bool;
    /// When the voice's note started, as a count of notes started so far.
    fn started(&self) -> u64;
    /// Whether the voice's note has ended and it is only fading out.
    fn is_releasing(&self) -> bool;
}

fn pick_voice<V: Voice>(voices: &mut [V]) -> &mut V {
    let oldest = |releasing: bool| {
        voices
            .iter()
            .enumerate()
            .filter(|(_, voice)| !releasing || voice.is_releasing())
            .min_by_key(|(_, voice)| voice.started())
            .map(|(index, _)| index)
    };
    let index = voices
        .iter()
        .position(Voice::is_idle)
        .or_else(|| oldest(true))
        .or_else(|| oldest(false))
        .unwrap_or(0);
    &mut voices[index]
}

```

The order is: a silent voice if there is one; otherwise the oldest voice that is only fading out; otherwise the oldest of all. Taking a busy voice is **voice stealing**, and every hardware synth does it when more notes are held than it has voices. A fading voice goes first because cutting a tail is far less noticeable than cutting a note still held, such as a long bass note under quick chords (the test `voices_that_are_fading_out_are_stolen_first` plays exactly that).

`Voice` is a private trait: `SamplerVoice` and `SynthVoice` are different structs, both implement it, and one generic function serves both. `V: Voice` is the trait bound; the function may only call what the trait promises.

Why find an index first and only then return `&mut voices[index]`? The searches borrow the slice to read it (`iter()`). A mutable reference can't come out of a read-only search, so the code finds the position, lets the read borrows end, and borrows mutably once at the end.

`started` comes from a counter that goes up with every note, so “oldest” is “smallest”. A `u64` won't overflow: at a thousand notes a second it would take 584 million years.

A fixed number of voices, in a box

```
voices: Box::new(std::array::from_fn(|_| {  
    SynthVoice::new(sample_rate, timbre.cutoff)  
})),
```

The voices are a fixed-size array, so the audio thread never grows it and never allocates. `std::array::from_fn` builds an array by calling the closure once per element; the closure gets the index, which is not needed here, hence `|_|`. The sampler's voices implement `Default`, so `Box::default()` builds them.

The array is in a `Box` because of the command queue to the audio thread. Every slot of the queue is as big as the biggest `Command`, and a command that inserts a channel carries an `Instrument`. Sixteen voices are over a kilobyte; a `Box` is one 8-byte pointer. A test pins the size down:

```
let size = std::mem::size_of::<Command>();  
assert!(size <= 80, "a command takes {size} bytes");
```

Boxing means the voices live on the heap, and freeing them is freeing memory. That is why an instrument the audio thread is done with always goes back to the UI thread in the trash queue ([Two threads](#)): the box is made and freed on the UI thread, and the audio thread only moves the pointer.

The Lineup Osc

The Lineup Osc plays one oscillator per voice, through a soft low-pass filter, shaped by a volume **envelope**:

```
pub struct Synth {  
    sample_rate: f32,  
    voices: Box<[SynthVoice; SYNTH_VOICES]>,  
    /// Boxed, like the voices, so an `Instrument` stays small.  
    envelope: Box<EnvelopeRates>,  
    notes_started: u64,  
}
```

The wave

An oscillator keeps a **phase**, from 0 to 1 through one cycle, and the wave says what each phase sounds like:

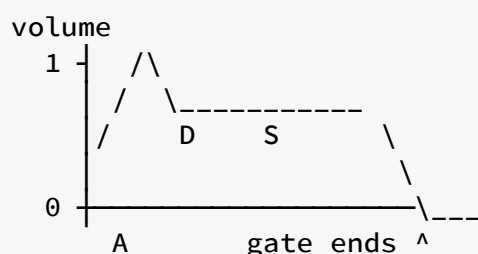
```
pub fn at(self, phase: f32) -> f32 {
    match self {
        // Ramps from -1 to 1.
        Wave::Saw => phase * 2.0 - 1.0,
        // 1 for the first half of the cycle, -1 for the second.
        Wave::Square => {
            if phase < 0.5 {
                1.0
            } else {
                -1.0
            }
        }
        // Up from -1 to 1 and back down.
        Wave::Triangle => 1.0 - 4.0 * (phase - 0.5).abs(),
        Wave::Sine => (phase * std::f32::consts::TAU).sin(),
    }
}
```

Each frame, the phase moves on by `frequency / sample_rate` and wraps round with `fract()`. A sawtooth has every harmonic, so it is bright and buzzy; a square only the odd ones, which sounds hollow; a triangle has weak ones and sounds soft; a sine has none and sounds pure.

A plain sawtooth made like this contains frequencies above what the sample rate can hold, and they fold back as faint metallic tones, which is called **aliasing**. The low-pass filter hides most of it. Removing it properly (with “PolyBLEP” corrections, or wavetables) would be a good project.

The envelope: a state machine

A note that just switches on and off clicks, and sounds like nothing real. An envelope shapes its volume over time. The classic one has four parts, **ADSR**:



A: attack, rising to full volume
D: decay, falling to...
S: sustain, held while the note is held
R: release, fading out after the note ends

Each voice is in one **stage**, and every sample may move it to the next. Code shaped like this is a **state machine**:

```

match self.stage {
  Stage::Idle => return 0.0,
  Stage::Attack => {
    self.level += envelope.attack_step;
    if self.level >= 1.0 {
      self.level = 1.0;
      self.stage = Stage::Decay;
    }
  }
  Stage::Decay => {
    let sustain = envelope.sustain;
    self.level = sustain + (self.level - sustain) *
envelope.decay_factor;
  }
  Stage::Release => {
    self.level *= envelope.release_factor;
    if self.level < SILENT {
      self.stage = Stage::Idle;
      return 0.0;
    }
  }
}

// Count down the gate while the note is held, then let go. A
// `HELD` gate is not counted: only `note_off` ends it.
if self.is_held() && self.gate != HELD {
  match self.gate.checked_sub(1) {
    Some(left) => self.gate = left,
    None => self.stage = Stage::Release,
  }
}

```

- The attack rises in a straight line; decay and release are **exponential**: each frame keeps a fixed fraction of the distance left, the way strings and bells fade. The fractions are worked out once per instrument in `EnvelopeRates::new`, not on every sample: `decay_factor` is $e^{(-1 / (\text{seconds} \times \text{sample_rate}))}$.
- `checked_sub` returns `None` instead of going below zero, so it reads as “if there is no gate left, release”.
- There is no `Sustain` stage. Decay approaches the sustain level and stays there, so a stage of its own would do nothing.
- `SILENT` is 0.0001, about -80 dB. An exponential fade never reaches zero, so the voice is called silent below it and freed.

`note_off` releases the voices of a pitch only if they were started with the `HELD` gate. A key let go on a MIDI keyboard must not end the same note that the song is playing on its own gate. `release_all` moves every held voice to `Release`, which is how Stop lets long notes fade instead of playing on.

A stolen voice is not reset: `start` sets the new pitch, gain and gate, and leaves the level, the phase and the filter as they were.

```
voice.stage = Stage::Attack;
voice.started = self.notes_started;
voice.origin = origin;
// The level, phase and filter carry on from where they were, so a
// stolen voice does not jump (and click).
```

The attack then rises from wherever the level was. Each voice is also turned down to `SYNTH_VOICE_GAIN`, 0.35, so a four-note chord does not come out four times too loud.

Timbres: sounding like a General MIDI instrument

The Lineup Osc can never sound like a real violin, but it can sound more like a violin than like a piano. A channel with a General MIDI program and no SoundFont plays a **timbre** chosen by the program's family:

```
pub struct Timbre {
    pub wave: Wave,
    /// Seconds to reach full volume.
    pub attack: f32,
    /// How quickly the volume falls to the sustain level (time constant,
    /// in seconds).
    pub decay: f32,
    /// The volume a held note settles at, 0 to 1. 0 for a sound that dies
    /// away by itself, like a piano or a plucked string.
    pub sustain: f32,
    /// How quickly a note fades after it is let go (time constant).
    pub release: f32,
    /// The low-pass filter's cutoff, in Hz: lower is darker.
    pub cutoff: f32,
    /// How loud, compared with the Lineup Osc's own sound.
    pub gain: f32,
}
```

Two small `const fn`s name the two ways instruments behave:

```
/// Struck or plucked: starts at once and dies away over `decay`.
const fn struck(wave: Wave, decay: f32, release: f32) -> Timbre {
    Timbre::new(wave, 0.003, decay, 0.0, release)
}

/// Bowed, blown or sung: swells in over `attack` and keeps going.
const fn held(wave: Wave, attack: f32, release: f32) -> Timbre {
    Timbre::new(wave, attack, 0.3, 0.85, release)
}
```

and `for_program` is one `match` on the program number, in General MIDI's own order, where each family is a range of eight numbers:

```
let timbre = match program.min(127) {
  // Pianos.
  0..=3 => Timbre::struck(Wave::Saw, 0.9, 0.25).cutoff(1_800.0),
  // ...
  // Brass.
  56..=63 => Timbre::held(Wave::Saw, 0.04, 0.1).cutoff(2_000.0),
  // Reeds: saxophones, oboe, bassoon, clarinet.
  64..=71 => Timbre::held(Wave::Square, 0.03, 0.08).cutoff(1_800.0),
  // Pipes: flutes, recorder, ocarina.
  72..=79 => Timbre::held(Wave::Triangle, 0.05, 0.1).cutoff(3_000.0),
  // ...
  // Sound effects have no tune to play; the Lineup Osc stands in.
  _ => Timbre::LINEUP_OSC,
};
```

Family	Wave	Envelope	Heard as
Piano, guitar, harp, pizzicato	sawtooth or triangle	starts at once, dies away even while held	struck or plucked
Strings, brass	sawtooth	swells in, keeps going	bowed or blown
Organ, accordion, reeds	square	full while held	hollow, reedy
Flutes, voices	triangle	soft attack, keeps going	soft and round
Bells, vibraphone, marimba	sine	dies away	pure

Range patterns (`0..=3`), alternatives (`24..=28 | 31`) and a final `_` make the whole table one expression, and the compiler checks that every number from 0 to 255 lands somewhere. At the end, each timbre's gain is multiplied by the wave's, so a square, the loudest wave, doesn't jump out of the mix. `Timbre::for_program(None)` is `LINEUP_OSC`, the synth's own bright sawtooth, so a channel without a program sounds as it always did. Which program each channel gets is in [SoundFonts and General MIDI](#).

Testing it

The tests play an instrument into a buffer and look at the numbers, since nobody can listen in a test:

- `synth_plays_the_right_pitch` plays A4 for a second and counts how often the output changes sign: about 880 times, two per cycle of 440 Hz.
- `synth_stops_after_the_gate_and_release` holds a note for a tenth of a second and checks the output is exactly zero ten release times later, with no voice playing.
- `synth_plays_chords`, `voices_are_stolen_when_all_are_busy` and `voices_that_are_fading_out_are_stolen_first` check the voices.
- `sampler_resamples_to_the_output_rate` plays one second recorded at 24 kHz on a 48 kHz output and expects 48 000 frames of sound, give or take ten; `sampler_plays_higher_notes_faster` expects half as many an octave up; `sampler_hits_can_overlap` plays two hits at once.
- `held_notes_last_until_note_off` checks the `HELD` gate.
- In `timbre.rs`, `plucked_instruments_die_away_and_bowed_ones_keep_going` checks the families, and `every_wave_stays_between_minus_one_and_one` the waves.

SoundFonts and General MIDI

Files: `crates/lineup-core/src/general_midi.rs`, `crates/lineup-audio/src/soundfont.rs`, `crates/lineup-audio/src/instrument.rs` and `crates/lineup-app/src/app/sounds.rs`

A channel says which instrument it plays with a number: 0 is a grand piano, 40 a violin, 52 a choir. The numbers come from **General MIDI**, a list every synthesizer agrees on. Three things in Ferrous Lineup turn such a number into sound:

- the Lineup Osc picks a wave and an envelope that come close to it ([Timbres](#));
- an external synth over MIDI is told to play it with a *program change* ([MIDI](#));
- a **SoundFont**, a file of recordings of real instruments, plays it inside the app.

This chapter is about the numbers, where a channel gets one, and the SoundFont. How to turn the sounds on and choose instruments in the app is in the manual's [Sounds](#).

General MIDI: 128 instruments by number

A MIDI cable or a MIDI file carries notes, not sounds. To say which sound should play them, MIDI has the program change message: “from now on, channel 3 plays program 40”. It is two bytes: `0xC2` (program change, `0xC0`, on channel 3, counted from 0 as 2) and `40`.

On its own, 40 means nothing. General MIDI, published in 1991, fixed a list of 128 programs, so that 40 is a violin on every synth that follows it, from a phone to a stage keyboard. The programs come in sixteen families of eight: pianos 0 to 7, chromatic percussion (celesta, vibraphone, marimba) 8 to 15, organs 16 to 23, guitars 24 to 31, basses 32 to 39, strings 40 to 47, and so on up to sound effects, 120 to 127. MIDI channel 10 is kept for drums: there, each *note* is a drum, 36 the kick, 38 the snare, 42 the closed hi-hat, whatever program is chosen. Manuals count programs from 1 (“41 Violin”); the bytes count from 0.

`general_midi.rs` in `lineup-core` holds the list. The programs the demos and styles use have names, and every program has its name and family:

```
pub const VIOLIN: u8 = 40;
pub const CELLO: u8 = 42;
pub const CHOIR_AAHS: u8 = 52;
```

```

/// The name of program `program`, such as "Violin" for 40.
pub fn name(program: u8) -> &'static str {
    NAMES[usize::from(program.min(LAST))]
}

/// The family program `program` belongs to, 0 (piano) to 15 (sound
effects).
pub fn family(program: u8) -> usize {
    usize::from(program.min(LAST)) / 8
}

```

`NAMES` is a `[&str; 128]` and `FAMILIES` a `[&str; 16]`, both in General MIDI's order. Because the families are eight programs each, "which family" is a division by 8, and the programs of family `f` are `f * 8..f * 8 + 8`. `.min(LAST)` keeps a number above 127 from indexing past the end: a `u8` can hold up to 255, and a damaged file could contain one. The menus are built from the same two arrays, and the names go through the translation table, so the Spanish menu says "Violín".

The built-in drums keep their own sounds, but anything that speaks General MIDI needs each one as a note of channel 10. `general_midi_drums`, in `midi_file.rs`, is that table:

```

/// The note General MIDI drum sets play for each built-in drum.
pub fn general_midi_drums(drum: Drum) -> Note {
    Note::new(match drum {
        Drum::Kick => 36,
        Drum::Snare => 38,
        Drum::Clap => 39,
        Drum::HiHat => 42,
        Drum::OpenHat => 46,
        Drum::Rim => 37,
        Drum::LowTom => 45,
        Drum::HighTom => 50,
        Drum::Ride => 51,
        Drum::Crash => 49,
        Drum::Cowbell => 56,
        Drum::Shaker => 70,
        Drum::Conga => 63,
    })
}

```

37 is the General MIDI "Side Stick", the rim shot; 70 is "Maracas", the closest it has to a shaker; 63 is "Open Hi Conga". `drum_name` goes the other way for the 47 drums from 35 to 81, so the mixer can say which drum a note plays. A test checks a few of them against the General MIDI list, and another checks the program names, so a typo in either table fails the tests.

A channel's instrument

`Channel` has one field for it:

```
/// The General MIDI instrument the channel plays, 0 to 127 (see
/// [general_midi`](crate::general_midi)), or None for none in
/// particular. MIDI files and the MIDI output send it to the synth as
/// a program change; the Lineup Osc picks a sound close to it.
#[serde(default, skip_serializing_if = "Option::is_none")]
pub program: Option<u8>,
```

`Option<u8>` because a channel can have no instrument in particular. Then nothing is sent to a synth, and the Lineup Osc keeps its own bright sawtooth. The two `serde` attributes keep old files readable: a file without the field reads as `None (default)`, and a channel without an instrument writes no field at all (`skip_serializing_if`). Only a project where some channel has an instrument needs format 9, so an older Ferrous Lineup still opens the rest ([The version number](#)).

A drum channel has no program: on MIDI channel 10 the notes choose the drums. A sampler playing a WAV file has none either, since its sound is the file.

Where the number comes from

- **The rack and the mixer.** Right-click a channel's name and open **Instrument**, or click the instrument's name in the mixer. Both show `instrument_choices`: "None (the Lineup Osc's own sound)", then the sixteen families, each a submenu of its eight programs, numbered from 1 as in the manuals. The family the current program is in has a dot before it, so you can find it in a long list.
- **The band.** The chord sheet's band panel gives each player an instrument with the same families ([How the band plays a chord sheet](#)).
- **The Library.** A style's **Keys** and **Bass** channels are made by `synth_named`, which gives them a piano and a bass:

```
let program = match name {
    KEYS => Some(general_midi::ACOUSTIC_GRAND_PIANO),
    BASS => Some(general_midi::FINGER_BASS),
    _ => None,
};
```

- **Demos.** Each voice of a demo names its instrument: violins and a cello for Pachelbel's Canon, a bandoneón (General MIDI's "Tango Accordion") for a tango ([How a demo is written](#)). A test builds every song demo and checks that every synth channel in it has a program.
- **MIDI files.** Importing a file takes each MIDI channel's first program change as its instrument ([MIDI](#)).

Where the number goes

1. **MIDI export** writes a program change at the very start of each channel's track, before its first note, so the synth is ready:

```
// The instrument first, so the synth is ready before the first note.
if let Some(program) = channel.program
    && midi_channels[index] != DRUM_CHANNEL
{
    let change = MidiMessage::ProgramChange {
        channel: midi_channels[index],
        program,
    };
    track.event(0, &change.bytes());
}
```

2. **MIDI output** sends it just before the channel's first note, and again after every stop, in case the sound was changed on the synth in between ([MIDI](#)).
3. **The Lineup Osc** plays a `Timbre` for the program's family.
4. **The SoundFont** plays the program's recordings, which is the rest of this chapter.

What a SoundFont is

A sampler plays one recording at different speeds ([The sampler](#)): twice as fast sounds an octave higher. That works for a few notes, but a piano recorded on middle C and played two octaves up sounds like a toy. The timbre of a real instrument changes across its range, and sped up, its attack gets too short.

A SoundFont (a `.sf2` file, a format from the sound cards of the 1990s) is a whole sampler library in one file:

- **samples:** short recordings, a piano's C, E, G# and so on every few notes, each with the note it was recorded at. Sounds that last, like an organ or strings, have a **loop**: once the attack has played, the middle of the recording repeats for as long as the key is held;
- **instruments:** which sample to play for which keys and velocities (a soft note can play a recording of a soft note), and how: the volume envelope, a filter, vibrato;
- **presets:** what a MIDI program change chooses, by **bank** and **program**.

A General MIDI SoundFont puts the 128 General MIDI instruments in bank 0, in their numbered order, and a drum kit in bank 128, which a synthesizer plays on MIDI channel 10. So program 0 is always a grand piano, program 33 a fingered bass guitar, and key 36 on channel 10 a kick.

Playing a note means finding the sample whose key and velocity range holds it, and playing it faster or slower by the distance between the two notes. Every semitone is a factor of $2^{(1/12)}$, about 1.0595, so a sample of A4 (440 Hz) played for B4, two semitones up, runs $2^{(2/12)} \approx 1.122$ times as fast: 494 Hz.

Who reads and plays it: rustysynth

Reading an `.sf2` file and playing it is a lot of code: parsing the file, the key ranges, the envelopes, the filters, the loops, mixing dozens of voices. The `rustysynth` crate does all of it, in plain Rust with no C library to install, under the MIT licence, which a GPL program may use. Its interface is small; in short:

```
let sound_font = Arc::new(SoundFont::new(&mut file)?);
let mut synthesizer = Synthesizer::new(&sound_font, &settings)?;
synthesizer.process_midi_message(0, 0xC0, 33, 0); // channel 1: program 33
synthesizer.note_on(0, 40, 100); // E2, velocity 100
synthesizer.render(&mut left, &mut right); // the next block of sound
```

`soundfont::load` is the first line with the errors spelled out: a file that cannot be opened is a `SoundFontError::Io`, and one that is not a SoundFont, or is damaged, is a `SoundFontError::Invalid`. Each prints a translated message, so the status bar can say “not a SoundFont file (...)” in the app’s language.

It speaks MIDI. And the app already has an instrument that speaks MIDI: the **MIDI Out**, which keeps track of which notes are sounding and when each one must end, and collects note-on, note-off and program-change messages in an outbox for an external synth ([MIDI](#)). So a SoundFont instrument is a MIDI Out whose messages go to a synthesizer inside the app.

A MIDI Out that plays inside the app

```
/// A channel's instrument played from a SoundFont.
pub struct SoundFontPlayer {
    /// Which notes sound and until when, as for an external synth; its
    /// messages go to the synthesizer once per block.
    notes: MidiOut,
    /// Boxed so an `Instrument` stays small.
    block: Box<Block>,
}

/// The synthesizer and the block of sound it rendered last.
struct Block {
    synthesizer: Synthesizer,
    sound_font: Arc<SoundFont>,
    /// [ `GAIN` ] or [ `DRUM_GAIN` ].
    gain: f32,
    left: [f32; BLOCK],
    right: [f32; BLOCK],
    /// The next frame of `left` and `right` to hand out. At `BLOCK`, a
    /// new block is due.
    read: usize,
}
```

The audio thread asks every instrument for one sample at a time. The synthesizer works in blocks of `BLOCK`, 64 frames, so the player hands out the block it has, and when the block runs out it passes on the messages of the last 64 frames and asks for the next block:

```
pub fn next_sample(&mut self) -> f32 {
    self.notes.next_sample();
    let block = &mut *self.block;
    if block.read == BLOCK {
        let synthesizer = &mut block.synthesizer;
        self.notes.flush(|message| play(synthesizer, message));
        synthesizer.render(&mut block.left, &mut block.right);
        block.read = 0;
    }
    let sample = (block.left[block.read] + block.right[block.read]) * 0.5
    * block.gain;
    block.read += 1;
    sample
}
```

`self.notes.next_sample()` makes no sound: a MIDI Out's `next_sample` counts frames, so it knows when each note's length is up and posts its note-off. A note that starts in the middle of a block is heard from the next one: at most 64 frames late, 1.3 ms at 48 kHz, far below what anyone hears.

`let block = &mut *self.block;` reaches through the `Box` once, so the lines after it borrow `block.synthesizer` and `block.left` as separate fields. Inside the `if`, `self.notes` and `block.synthesizer` are borrowed at the same time, one by `flush` and one by the closure; Rust allows it because they are different fields.

The SoundFont's sounds are in stereo, and every channel of the app is mono until the mixer places it with its pan ([The mixer](#)). So the two sides are added and halved.

`flush` is the same method that hands a MIDI Out's messages to the MIDI port, and `play` turns each `MidiMessage` into the synthesizer's call:

```
/// Hands one MIDI message to the synthesizer.
fn play(synthesizer: &mut Synthesizer, message: MidiMessage) {
    match message {
        MidiMessage::NoteOn {
            channel,
            pitch,
            velocity,
        } => synthesizer.note_on(
            i32::from(channel),
            i32::from(pitch.midi()),
            i32::from(velocity),
        ),
        MidiMessage::NoteOff { channel, pitch } => {
            synthesizer.note_off(i32::from(channel),
            i32::from(pitch.midi()))
        }
        MidiMessage::ProgramChange { channel, program } => {
            synthesizer.process_midi_message(i32::from(channel), 0xC0,
            i32::from(program), 0)
        }
    }
}
```

`i32::from` widens a `u8` to the `i32` `rustsynth` takes; it cannot fail, which is why it is `from` and not `try_from`.

Everything a MIDI Out already does right comes for free: notes that end after their length, held keys from the keyboard, notes that stop when their track is muted, note lengths that stretch with a tempo change. The engine also treats both alike in one more way: MIDI has one of each note per channel, so a note that starts again ends the one sounding (`one_per_pitch` in `engine.rs`).

A player is made for a program:


```
pub fn new(sound_font: &Arc<SoundFont>, sample_rate: u32, program: u8) ->
Option<Self> {
    let notes = MidiOut::new(0).with_program(Some(program));
    Self::on(sound_font, sample_rate, notes, GAIN)
}
```

`with_program` makes the MIDI Out send the program change before its first note, so the synthesizer is on the right instrument when the first note arrives. A drum is a MIDI Out on channel 10 that plays one note whatever note its steps are on (`playing_as`), as a drum sent to an external synth does:

```
pub fn drum(sound_font: &Arc<SoundFont>, sample_rate: u32, drum: Note) ->
Option<Self> {
    let notes = MidiOut::new(DRUM_CHANNEL).playing_as(Some(drum));
    Self::on(sound_font, sample_rate, notes, DRUM_GAIN)
}
```

One synthesizer per channel, one SoundFont for all

A synthesizer can play sixteen MIDI channels at once, so one could play the whole band. Each channel of the app gets its own instead. That way its volume, pan, effects, mute light and level meter work exactly as on any other channel, because the engine mixes channels, not MIDI channels.

The SoundFont itself is tens of megabytes, and it is not copied. It is read once into an `Arc<SoundFont>`, an *atomically reference-counted* pointer: `Arc::clone` copies the pointer and adds one to a counter, and the memory is freed when the last clone goes. Every synthesizer holds one clone, and so does the app. An instrument the audio thread replaces goes back on the trash queue and is dropped on the window's thread ([Freeing memory somewhere else](#)), so even when it holds the last clone, the SoundFont is never freed on the audio thread.

`Instrument` has a kind for it, and a constructor that says which channels the SoundFont plays:

```
pub fn from_sound_font(
    channel: &Channel,
    sample_rate: u32,
    sound_font: &Arc<SoundFont>,
    drums: bool,
) -> Option<Self> {
    let player = match &channel.kind {
        ChannelKind::Synth => SoundFontPlayer::new(sound_font,
sample_rate, channel.program?),
        ChannelKind::Sampler(SampleSource::Builtin(drum))
            if drums && channel.sent_on.is_none() =>
        {
            SoundFontPlayer::drum(sound_font, sample_rate,
general_midi_drum(*drum))
        }
        _ => None,
    };
    player.map(Instrument::SoundFont)
}
```

The `?` after `channel.program` returns `None` from the whole function when the channel has no program: that is what keeps a channel on “None” playing the Lineup Osc. A built-in drum plays the SoundFont’s kit when the drums setting is on, unless it is being sent to an external synth (`sent_on`). A sample file and a MIDI Out are not the SoundFont’s to play. The app’s `build_instrument` asks this first and falls back on `Instrument::for_channel` when the answer is `None` :

```
fn build_instrument(&mut self, channel: &Channel, sample_rate: u32) ->
Instrument {
    if let Some((sound_font, drums)) = self.sounds.in_use()
        && let Some(instrument) =
            Instrument::from_sound_font(channel, sample_rate, sound_font,
drums)
    {
        return instrument;
    }
    match Instrument::for_channel(channel, sample_rate, &mut self.samples)
    {
        ...
    }
}
```

Every instrument is built there, for playing and for exporting, so **File > Export WAV** and the video export play the SoundFont too.

A synthesizer’s two blocks of 64 samples are half a kilobyte, and every slot of the queue to the audio thread is as big as the biggest command. That is why the `Block` is boxed, as the other instruments box their voices ([A fixed number of voices, in a box](#)); the test `commands_stay_small` would catch a command that grew.

Nothing allocated on the audio thread

The audio thread never asks for memory ([The golden rule](#)). A synthesizer reserves everything it needs when it is made: its voices (`VOICES` , 128 per channel; a piano note can use two, its left and right samples), their buffers, the table of presets.

`Synthesizer::new` runs on the window's thread, inside `build_instrument` , and the finished instrument is sent across. Its `note_on` and `render` only reuse those voices and buffers.

`on` sets the synthesizer up:

```
let mut settings =
  SynthesizerSettings::new(i32::try_from(sample_rate).ok()?);
settings.block_size = BLOCK;
settings.maximum_polyphony = VOICES;
// Reverb and chorus would cost as much as the voices, once for
// every channel. The sounds are recorded with some room already.
settings.enable_reverb_and_chorus = false;
let synthesizer = Synthesizer::new(sound_font, &settings).ok()?;
```

The mixer has one reverb that every channel sends into, so a room is still there, once ([The mixer](#)). `i32::try_from(sample_rate).ok()?` is the careful conversion: a `u32` above `i32::MAX` would not fit, and `?` on the `Option` gives up instead of wrapping around. The synthesizer refuses rates below 16 kHz and above 192 kHz itself, and then the channel keeps its Lineup Osc.

Loud enough

rustysynth plays quietly, at half volume, because it expects a whole band to share it. The player turns it up by `GAIN` (5) and the drums by `DRUM_GAIN` (7). Those numbers come from a measurement, not from taste: the same notes were rendered on the Lineup Osc and on the SoundFont, and their peaks and average levels compared. A real piano dies away while the Lineup Osc holds its notes, so the two never match exactly, and another SoundFont may be louder or softer; each channel's volume does the rest.

The offer, and the download

The app does not come with a SoundFont, because a good one is tens of megabytes. As soon as the app opens with its instruments on the Lineup Osc, before anything plays, a window offers one:



Download fetches GeneralUser GS, a free General MIDI SoundFont by S. Christian Collins, from its own repository, and keeps it in the app's data folder:

~/ .local/share/ferrouslineup/soundfonts on Linux, the Application Support folder on macOS, AppData\Roaming on Windows. **Choose a file...** takes any other .sf2. **Not now** keeps the Lineup Osc. **Edit > Settings... > Audio** has the same choices for later: **SoundFont instruments**, **SoundFont drums**, **Choose SoundFont...** and **Use GeneralUser GS**.

If GeneralUser GS is already in that folder, downloaded by an earlier run or an earlier version, there is nothing to download, and the window offers **Use it** instead:



Asked until answered

One method decides whether to ask:

```
pub(super) fn offer_sound_font(&mut self) {
    let sounds = &mut self.sounds;
    if !sounds.settings.on && !sounds.settings.turned_down &&
        sounds.general_user.is_some() {
        sounds.offer_open = true;
    }
}
```

It opens the window whenever the instruments play on the Lineup Osc and the offer was not turned down. Nothing is asked while the sounds are on, even if their SoundFont is still being read, nor on a system with no folder for app data (`general_user` is `None` there).

It is called in two places. `LineupApp::new` calls it once the app is made, so the question is the first thing the window shows:

```
// Real instrument sounds are offered as soon as the window opens,
// before anything plays.
app.offer_sound_font();
```

and every way of starting to play (the ► button, **Space**, a Library preview, opening a demo) goes through `set_playing`, which ends with

```
self.playing = true;
self.offer_sound_font();
```

so an offer that was closed unanswered comes back on Play. Playing goes on while the window asks, so the sound is never held up by a question.

A run that did not close cleanly has a question of its own at the start: whether to bring back the unsaved work (see [A copy of unsaved work](#)). The two would cover each other, so the sounds wait for it:

```
// Unsaved work from a run that did not close cleanly is offered
// first.
if !self.sounds.offer_open || !self.recovery.pending.is_empty() {
    return;
}
```

Opening the window answers nothing. A run that closes before the question is answered asks again as the app next opens, and so does a run whose SoundFont has gone missing. Only a “no” ends the offer:

```
fn decline_sound_font(&mut self) {
    let sounds = &mut self.sounds;
    sounds.offer_open = false;
    sounds.settings.turned_down = true;
    sounds.turn_off();
}
```

Not now calls it, and so do closing the window before a download and **Stop** during one. Turning **SoundFont instruments** off in Settings sets `turned_down` too, and turning it on clears it. A “yes” turns the sounds on and clears `turned_down`:

```
// Yes: GeneralUser GS, even in place of a chosen file that is
// gone, played from as soon as it is downloaded or read.
if (download || use_it)
    && let Some(path) = self.sounds.general_user.clone()
{
    let sounds = &mut self.sounds;
    sounds.settings.file = None;
    sounds.settings.turned_down = false;
    if download {
        sounds.settings.on = true;
        sounds.changed = true;
        sounds.start(path, true);
    } else {
        sounds.offer_open = false;
        self.set_sound_font(true);
    }
}
```

Download starts the download and leaves the window open to show it. **Use it** closes the window and goes through `set_sound_font`, the same as ticking the setting, which finds the file on disk and starts reading it. Either way, when the SoundFont is ready, `check_sounds` gives every channel its instrument again and the status bar says “Instruments play from GeneralUser GS”. The window is drawn in egui’s `Foreground` order, above every other window, because a demo opens its chord sheet and brings it to the front in the same frame it starts to play.

The settings are a small struct:

```
#[derive(Debug, Clone, PartialEq, Serialize, Deserialize)]
#[serde(default)]
struct Settings {
    /// Whether channels play from the SoundFont.
    on: bool,
    /// Whether the built-in drums do too.
    drums: bool,
    /// A SoundFont file chosen with "Choose SoundFont...", or `None` for
    /// GeneralUser GS.
    file: Option<PathBuf>,
    /// Whether the offer of real instrument sounds was turned down, with
    /// "Not now", or the sounds were turned off in Settings. Until then,
    /// the app offers them as it opens and on Play whenever they are off.
    turned_down: bool,
}
```

They are kept between runs by name, as `"sound-font"`, with the app’s `preferences` module ([Kept by name, not by type](#)):

```
settings: preferences::get(ctx, SETTINGS_NAME).unwrap_or_default(),
```

when the app starts, and `preferences::set(ctx, SETTINGS_NAME, &self.sounds.settings)` whenever they change. Kept by name, they survive an update of the app. In egui's own memory they would not: an update would turn the sounds off, and with GeneralUser GS still on disk the app would be back on the Lineup Osc.

`#[serde(default)]` on the struct fills any field missing from the saved data with its value from `Default`, so a field can be added without losing everyone's settings, and a field the struct no longer has is skipped. Renaming a field therefore forgets only that field: a renamed `turned_down` starts as `false`, and everyone is asked again. `Default` is written by hand rather than derived, because `drums` starts as `true`.

Downloading on a thread of its own

A download of 30 MB takes from a second to a minute, and reading a SoundFont takes a moment too. Both run on a thread of their own, like the WAV export, and the window keeps drawing. The thread and the window share a `Progress`:

```
/// How far a download is. Its thread writes it and the UI thread reads
/// it; atomics need no lock for that.
#[derive(Default)]
struct Progress {
    /// Bytes downloaded so far.
    done: AtomicU64,
    /// The size of the file, or 0 while it is not known.
    total: AtomicU64,
    /// Set by the UI thread to stop the download.
    cancel: AtomicBool,
}
```

Atomics are numbers two threads can read and write at the same time without a lock: the download stores how many bytes it has, the window reads it to draw the bar, and **Stop** stores `true` in `cancel`, which the download checks before each read of up to 64 KB. `Ordering::Relaxed` is enough, because no other data hangs on these numbers: a bar one frame behind is fine.



The request is made with `ureq`, a small HTTP client, using the certificates the operating system trusts (so it works behind a company proxy that has its own) and the proxy in `HTTPS_PROXY`, if there is one. The environment variable `FERROUS_LINEUP_SOUNDFONT_URL` gives another address, such as a copy on your own network. A file over a gigabyte is refused: no General MIDI SoundFont is that big, so it is something else.

The file is written as `GeneralUser-GS.sf2.part`, then read as a SoundFont, which checks it is one, and only then renamed:

```
fn download_to(url: &str, path: &Path, progress: &Progress) ->
Result<Arc<SoundFont>, String> {
    let part = path.with_extension("sf2.part");
    let result = download_part(url, &part, progress).and_then(|()| {
        let sound_font = soundfont::load(&part).map_err(|err|
err.to_string());
        fs::rename(&part, path).map_err(|err| err.to_string());
        Ok(sound_font)
    });
    if result.is_err() {
        let _ = fs::remove_file(&part);
    }
    result
}
```

A download cut off halfway, or a web page that came back instead of the file, never leaves a broken `GeneralUser-GS.sf2` that would fail every time the app starts.

`and_then` runs the second step only if the first worked, and `let _ =` says on purpose that a `.part` file that cannot be removed is not worth reporting.

Once per frame, `check_sounds` stores the settings if they changed and asks the thread whether it is done (`JoinHandle::is_finished`). While it is not, it asks egui to draw again in a tenth of a second, so the bar moves even when the mouse does not. When it is, `join` hands over the `Arc<SoundFont>`, and every channel needs a new instrument made from it. The app already knows how to do that: when channels start or stop being sent over MIDI, `send_routes` rebuilds the instruments whose route changed. Forgetting all the routes makes it rebuild every one on the next frame:

```
/// Has every channel of the audio thread given its instrument again,  
/// on the next frame: the SoundFont was turned on or off.  
pub(super) fn instruments_changed(&mut self) {  
    self.routes_sent = None;  
}
```

Turning the sounds off does the same, and every channel gets its Lineup Osc or its built-in drum back. The SoundFont stays loaded while the sounds are off, so turning them on again is instant.

Debug builds

`cargo run` builds without optimizations, and a synthesizer works out every sample of every voice: unoptimized, a band could be too much for the audio thread. Cargo can optimize one dependency while the rest stays quick to build, in the workspace's

Cargo.toml :

```
[profile.dev.package.rustysynth]  
opt-level = 3
```

Testing it

The tests should not need a 30 MB download, and the repository should not hold one. So the test module writes a SoundFont byte by byte: one sample, a sine wave at 440 Hz that loops, two instruments (one plays the sample as recorded, one an octave higher) and three presets (program 0, program 33, and a drum kit in bank 128).

An `.sf2` file is a RIFF file, the same container as a WAV file: a tree of *chunks*, each a four-letter name, a size and its bytes. Two small functions build them:

```
/// One chunk of a RIFF file: its name, its size and its bytes, padded
/// to an even length.
fn chunk(id: &[u8; 4], body: &[u8]) -> Vec<u8> {
    let mut bytes = id.to_vec();
    bytes.extend((body.len() as u32).to_le_bytes());
    bytes.extend(body);
    if body.len() % 2 == 1 {
        bytes.push(0);
    }
    bytes
}

/// A list of chunks: a chunk whose bytes are a type and other chunks.
fn list(kind: &[u8; 4], chunks: &[Vec<u8>]) -> Vec<u8> {
    let mut body = kind.to_vec();
    for each in chunks {
        body.extend(each);
    }
    chunk(b"LIST", &body)
}
```

The file is a list of three lists: `INFO` (a version and a name), `sdta` (the samples, as 16-bit numbers) and `pdta` (the presets, instruments and sample headers, in nine fixed-size tables). The tests then play it and listen the way an oscilloscope would: a sine wave crosses zero twice per period, so counting the crossings in a quarter of a second, times two, gives its frequency.

```
#[test]
fn the_program_picks_the_instrument() {
    let mut player = SoundFontPlayer::new(&tiny_sound_font(), RATE,
33).unwrap();
    player.start(Note::new(A4), 100, HELD, None);
    let sound = render(&mut player, RATE as usize / 2);
    assert!(
        (frequency(&sound) - 880.0).abs() <= 8.0,
        "{}",
        frequency(&sound)
    );
}
```

Program 33 is the instrument an octave up, so A4 comes out at 880 Hz. Other tests check that a player is silent until a note starts, that A3 comes out at 220 Hz, that a note starts within one block and ends after its length, that louder notes are louder, that a drum plays its own key whatever note it is written on, that a file that is not a SoundFont is refused, and which channels the SoundFont takes (`channels_without_a_sound_in_the_sound_font_keep_theirs`).

The window tests must not depend on whether the computer running them has downloaded GeneralUser GS. So every test app looks for it at a path that never exists, `keep_general_user_in(no_sound_font())`, and starts as if the offer had been turned down. Four tests undo that. `the_app_offers_real_instrument_sounds_as_it_opens` does what `LineupApp::new` does and checks that the window opens before anything plays, with **Download**, and that after **Not now** it does not come back, even after a restart and an update.

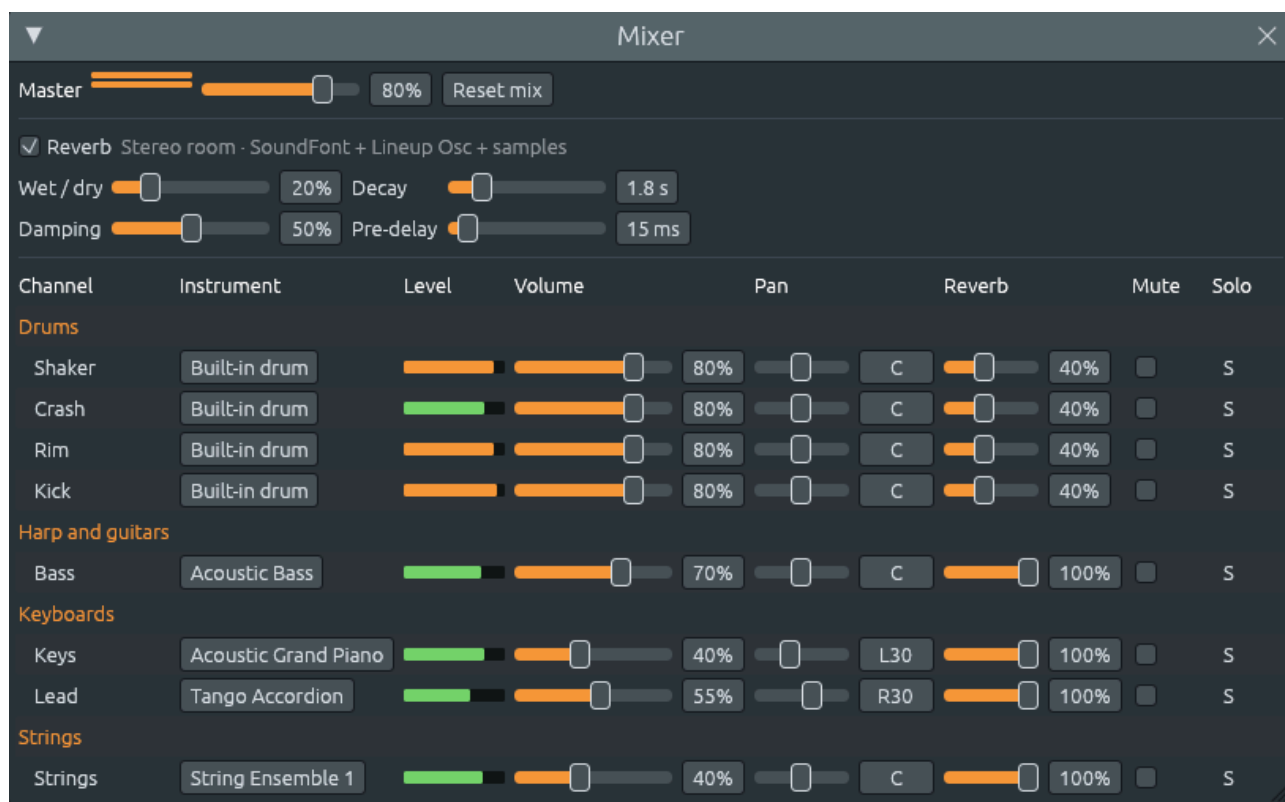
`play_offers_real_instrument_sounds_until_the_offer_is_answered` presses **Space** and checks that the window opens while the song plays, with **Download**. It starts the app again without answering and checks that Play asks again. Then it clicks **Not now**, presses Play twice, and checks that nothing is asked, even after a restart and after an update. `play_offers_a_sound_font_already_on_disk` puts an empty file where GeneralUser GS goes, checks that the window offers **Use it**, clicks it, and checks that the file is being read.

`a_demo_asks_for_real_instrument_sounds_in_front_of_its_chord_sheet` checks that the window is on top, and that its buttons take the click.

The mixer: levels, pan, reverb, effects and score order

Files: `crates/lineup-core/src/mixer.rs`, `crates/lineup-core/src/effects.rs`, `crates/lineup-core/src/score_order.rs`, `crates/lineup-audio/src/reverb.rs`, `crates/lineup-audio/src/effects.rs`, `crates/lineup-audio/src/levels.rs`, `crates/lineup-app/src/app/mixer.rs` and `crates/lineup-app/src/meters.rs`

The mixer (**F9**) is where the band is balanced. Each channel has a line: its instrument, a level meter, its volume, where it sits between the speakers (**Pan**), how much of it goes into the room (**Reverb**), its own **Effects** (an equalizer, a compressor and a delay), **Mute** and **Solo**. Above them are the master volume with a meter for each speaker, and the reverb that every channel shares. The lines are listed the way a conductor's score lists instruments, in sections.



The work is split as everywhere else in the program. `lineup-core` holds the settings, which are saved with the song; `lineup-audio` turns them into sound on the audio thread; `lineup-app` draws the window. How to use the mixer is in the manual's [The mixer](#).

What is saved

The project has one `Mixer`:

```
[derive(Debug, Clone, Copy, PartialEq, Serialize, Deserialize)]
[serde(default)]
pub struct Mixer {
    pub master: f32,
    pub reverb: ReverbSettings,
    pub solo: Option<usize>,
}
```

```
[derive(Debug, Clone, Copy, PartialEq, Serialize, Deserialize)]
[serde(default)]
pub struct ReverbSettings {
    pub enabled: bool,
    /// Wet fraction, from 0 to 1.
    pub mix: f32,
    /// Approximate time to decay by 60 dB, in seconds.
    pub decay: f32,
    /// High frequency absorption, from 0 to 1.
    pub damping: f32,
    pub pre_delay_ms: f32,
}
```

and each `channel` has its own mix: `volume` and `muted`, and three more fields:

```
[serde(default, skip_serializing_if = "is_centred")]
pub pan: f32,
[serde(default = "full_send", skip_serializing_if = "is_full_send")]
pub reverb_send: f32,
[serde(default, skip_serializing_if = "Effects::is_plain")]
pub effects: Effects,
```

`skip_serializing_if` leaves a field out of the file while it has the value of a new channel (in the middle, everything sent to the reverb, no effects), and `default` fills it back in when it is missing. A song whose channels are all plain is written exactly as it would be by a Ferrous Lineup with no pans or effects, so an older version still opens it. Only a song that uses them needs format 26 (pan and send) or 28 (effects) ([The version number](#)). Opening checks the ranges: a file edited by hand with a pan of -3, or a delay whose echoes would never die away, is refused with a message instead of playing a channel louder than full.

Solo is one channel number, not a flag on each channel. The project answers “is this channel heard?” in one place, so everything that plays (the audio thread, an export, a MIDI file) agrees:

```
pub fn channel_muted(&self, channel: usize) -> bool {
    self.channels.get(channel).is_none_or(|state| state.muted)
    || self.mixer.solo.is_some_and(|solo| solo != channel)
}
```

That is why soloing a channel reaches the audio thread as a mute change on every other channel ([diff: the list of changes](#)).

Reset mix puts the master, the reverb, the solo and every channel's volume, pan, send and effects back as they are on a new song. The band's players get their usual volumes back, and channels the band keeps quiet stay muted (`Project::reset_mix`, in `band.rs`).

The path of one sample

For each frame, the audio thread takes every channel through the same steps. This is the heart of the mix, in `engine.rs`:

```
let wet = self.reverb.wet();
let mut dry = [0.0; 2];
let mut send = 0.0;
for channel in &mut self.channels {
    let volume = channel.volume.next_value();
    let sample = channel.effects.process(channel.instrument.next_sample())
* volume;
    channel.peak = channel.peak.max(sample.abs());
    let amount = channel.reverb_send.next_value();
    let gains = lineup_core::Channel::pan_gains(channel.pan.next_value());
    let kept = sample * (1.0 - wet * amount);
    dry[0] += kept * gains[0];
    dry[1] += kept * gains[1];
    send += sample * amount;
}
// ...
let click = self.click.next_sample();
let stereo = self
    .reverb
    .next(dry, send, wet)
    .map(|sample| sample + click);
let volume = self.master_volume.next_value();
let stereo = stereo.map(|sample| sample * volume);
master_peak = [0, 1].map(|side|
    master_peak[side].max(stereo[side].abs()));
// `tanh` gently rounds off peaks above 1.0 instead of clipping them hard.
let stereo = stereo.map(f32::tanh);
```

In order:

1. The instrument makes a sample, mono.
2. The channel's **effects** shape it: equalizer, compressor, delay.
3. Its **volume** scales it, and its **peak** is kept for the meter.
4. Its **pan** splits it between the two speakers, and its **send** puts part of it into the reverb.
5. The **reverb** adds the room to the sum of all channels.
6. The metronome's click is added after the reverb: the room is the music's, and a click with a tail is harder to follow.
7. The **master volume** scales the whole, its peaks go to the master meters, and `tanh` rounds off anything above full scale ([Mixing a frame](#)).

Volume, pan, send and the reverb's wet level are all `Smoothed`: moved in the window, they glide to the new value over a few milliseconds instead of jumping, since a jump in the middle of a wave is heard as a click. The effects come before the volume, as on a mixing desk's channel strip, so a compressor hears the instrument as it plays, whatever the fader says.

Pan: two speakers, one channel

Pan is a number from -1 (all left) to 1 (all right), with 0 in the middle. It becomes a gain for each speaker:

```
/// How loud it is in the left and the right speaker for its pan: both
/// full in the middle, and the far side fading out as it moves away.
pub fn pan_gains(pan: f32) -> [f32; 2] {
    [(1.0 - pan).min(1.0), (1.0 + pan).min(1.0)]
}
```

pan	-1.0	-0.5	0.0	0.5	1.0
left	1.0	1.0	1.0	0.5	0.0
right	0.0	0.5	1.0	1.0	1.0

In the middle both speakers play the channel at full volume, the way a mono sound plays on a stereo system. Moving a channel to one side turns the **other** speaker down, like the balance knob of a stereo.

This is one of several **pan laws**. Mixing desks often use a *constant-power* law instead, where the middle plays each speaker at about 0.71 (-3 dB), so a channel sounds equally loud wherever it is placed. With the law used here, a channel moved hard left sounds a little quieter than in the middle, because only one speaker plays it. The

reason for keeping it is that a song where nothing is panned plays every channel at exactly its volume on both sides.

The slider writes its value as `L30`, `C` or `R100`, and reads what you type the same way: `pan_value` accepts `L30`, `R100`, `C`, a plain number from `-100` to `100`, and the Spanish `I30` (*izquierda*) and `D30` (*derecha*), which is what the Spanish interface shows. A double-click puts it back in the middle. A slider only senses drags, so the double-click is read from the pointer and checked against the slider's rectangle.

The reverb: a room made of delays

A reverb imitates a room. A sound bounces off the walls, the floor and the ceiling and comes back, later and softer, thousands of times, until the echoes melt into a tail that dies away over a second or two. The mixer's reverb is a **Schroeder reverb**, the oldest digital design (Manfred Schroeder, 1962), built from two kinds of delay.

A **comb filter** is a delay whose output is fed back into its input, a little softer each time: a sound comes back every 31 ms, say, quieter on each lap. One comb sounds like a metallic flutter, but four in parallel, with lengths that never line up, give a dense pattern of echoes. An **all-pass filter** is a delay that smears each echo into a cluster without changing the overall tone; three in a row turn the combs' echoes into a smooth wash.

```
fn new(rate: u32, spread: usize) -> Self {
    let length =
        |samples| ((samples + spread) as f32 * rate as f32 /
48_000.0).round() as usize;
    Self {
        combs: [1493, 1601, 1747, 1867].map(|samples| Comb {
            delay: Delay::new(length(samples)),
            feedback: 0.0,
            filtered: 0.0,
        }),
        diffusers: [241, 503, 719].map(|samples|
Delay::new(length(samples))),
    }
}
```

The lengths are counted in samples at 48 kHz (1493 samples is 31 ms) and scaled to the actual rate. They are all prime numbers, so no two share a factor and their echoes rarely fall on the same instant, which would sound like a ringing note. Each side of the stereo room is a `Side` with its own combs and all-passes; the right one is made with

`spread 37`, so its delays are 37 samples longer and the two sides differ, which the ear hears as width.

`[...].map(...)` on an array gives an array of the same length, so `combs` is a `[Comb; 4]` built in one expression, with no `Vec`.

How long the room rings

Decay is the time the tail takes to fall by 60 dB, to a thousandth of its level: the usual measure of a room, called RT60. Each comb's feedback is worked out from it:

```
fn configure(&mut self, rate: u32, decay: f32) {
    for comb in &mut self.combs {
        comb.feedback =
            10.0_f32.powf(-3.0 * comb.delay.samples.len() as f32 / (rate
as f32 * decay));
    }
}
```

In `decay` seconds the sound goes round a comb `rate × decay / length` times, and is multiplied by `feedback` each time. For the total to be 10^{-3} (−60 dB), each lap multiplies by $10^{-(3 \times \text{length} / (\text{rate} \times \text{decay}))}$. Longer combs go round fewer times, so they get a stronger feedback, and all four die away together.

Damping imitates the air and the soft walls of a real room, which swallow high frequencies sooner than low ones. Inside each comb, the echo goes through a one-pole low-pass filter on its way back in:

```
fn next(&mut self, input: f32, damping: f32) -> f32 {
    let out = self.delay.read();
    self.filtered = flushed(out * (1.0 - damping) + self.filtered *
damping);
    self.delay.write(input + self.filtered * self.feedback);
    out
}
```

`filtered` moves only part of the way towards each new sample, which smooths out fast wiggles: the highs. Each lap smooths them a little more, so the tail gets darker as it fades, as in a hall.

Pre-delay is the gap before the room answers, up to 200 ms: in a big hall the first reflection arrives later than in a small room. It is one more ring buffer in front of the combs, a fifth of a second long.

Never too small, never in the way

`flushed` turns any value under 10^{-20} into exactly 0. A fading tail gets smaller and smaller without ever reaching zero, and numbers that tiny (**denormals**) make many processors dozens of times slower. A reverb left to ring out in silence could then overrun the audio thread's time.

All delay memory is reserved in `Reverb::new`, before the stream starts; `configure` only changes numbers, so moving a slider never allocates on the audio thread. A setting out of range is ignored rather than applied. Turned off, the wet level fades to zero and then `Reverb::next` returns the dry sound at once, doing no work. Turned on again, it starts from silence (`clear`), not from the tail it had when it was switched off.

The send: how much of each channel is in the room

Each channel has a **send**, from 0 to 1: how much of it goes into the room. The sum of all sends is the reverb's single input, so twenty channels cost one reverb, not twenty, as with a send on a real desk. The reverb returns only the room, which `next` adds to the dry sound:

```
[
    dry[0] + left.next(delayed, damping) * wet,
    dry[1] + right.next(delayed, damping) * wet,
]
```

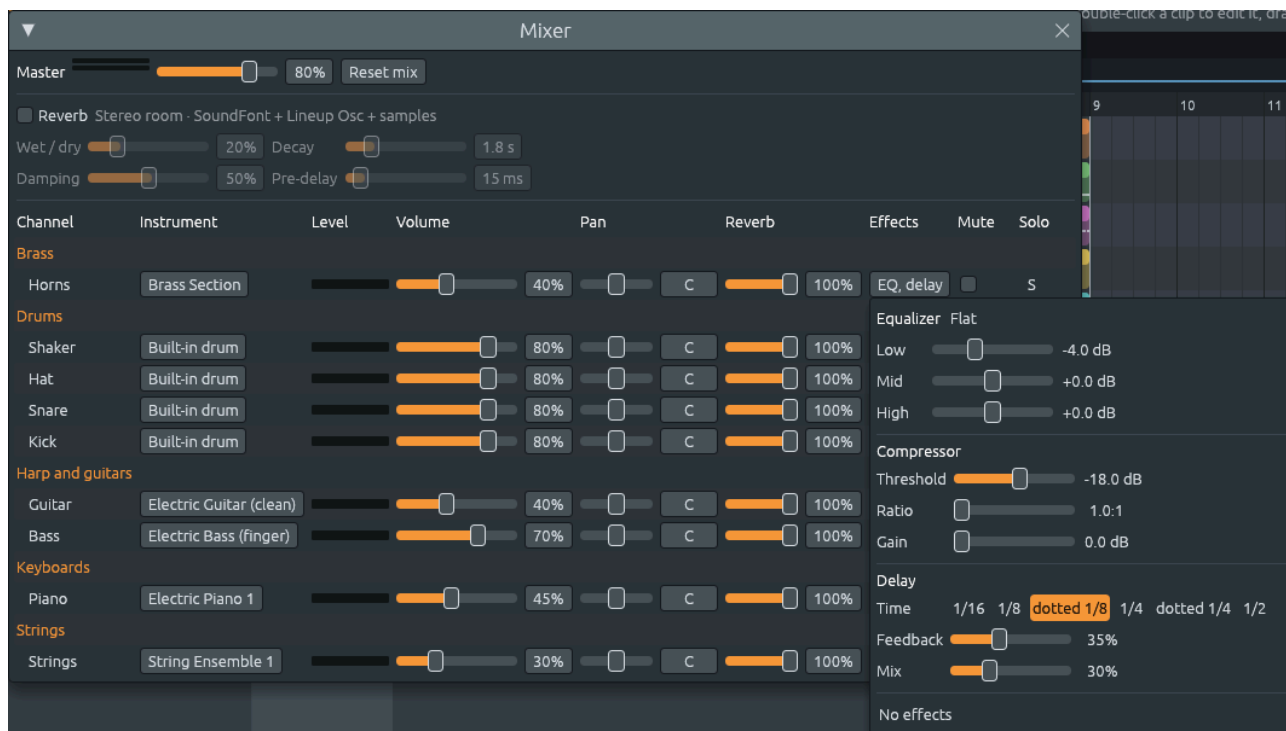
The line to look at in the mix is `kept = sample * (1.0 - wet * amount)`: what goes into the room is taken out of the dry sound, in proportion to the **Wet / dry** setting. Check the two ends:

- **Send 1** (all of it, as on every new channel): `kept = sample * (1 - wet)`, and the whole sample goes to the room. That is a classic wet / dry mix of the whole song.
- **Send 0**: `kept = sample`, and nothing goes to the room. The channel plays dry and close, as with the reverb off, while the rest of the band stays in the room.

The room is mono going in and stereo coming out, so a channel panned hard left still has its reverb on both sides, as a guitarist on the left of a stage fills the whole hall.

Each channel's effects

A real mixing desk has, on every channel strip, a few effects that shape one instrument before it joins the others. Each mixer line has three, in the order most desks have them: an **equalizer**, a **compressor** and a **delay**.



The equalizer: turning frequencies up or down

Every sound is a mix of frequencies: a note's fundamental, and its overtones above it. An **equalizer** (EQ) turns ranges of them up or down. It adds no notes; it changes the colour of the ones there. The mixer's EQ has three bands, each from -12 dB to +12 dB:

Band	Where	Shape	What lives there
Low	below 200 Hz	a shelf	the weight of a bass, a kick drum, a piano's left hand
Mid	around 1 kHz	a bell, about an octave and a half wide	the body of most instruments, and of voices
High	above 4 kHz	a shelf	the air and sparkle: cymbals, a breathy voice, a bow on a string

A **shelf** turns everything past its frequency up or down by the same amount, like a step. A **bell** turns its frequency up or down most, and less and less on each side.

Decibels measure a ratio: +6 dB makes a band's waves about twice as tall, -6 dB about half, 0 dB leaves it as it is.

Turning things **down** is the commonest use. When two instruments play in the same range, they mask each other: a piano's lows and the bass turn into one muddy sound. Taking the lows out of the piano leaves room for the bass, and once the bass plays, the piano sounds no thinner.

On the audio thread each band is a **biquad**, a second-order filter, the building block of most equalizers. Each sample out is a mix of the last three samples in and the last two out:

```
let output = b0 * input + b1 * self.inputs[0] + b2 * self.inputs[1]
    - a1 * self.outputs[0]
    - a2 * self.outputs[1];
```

The five numbers `b0`, `b1`, `b2`, `a1` and `a2` decide what the filter does. `Biquad::set` works them out for a shelf or a bell, a frequency and a gain, with the formulas of Robert Bristow-Johnson's *Audio EQ Cookbook*, which almost every audio program uses: a slope of 1 for the shelves, a Q of 1 for the bell. The three filters run one after the other, low, mid, high, and their numbers are only worked out again when the settings change, never for each sample. `set` keeps the samples the filter remembers, so moving a band while it plays does not click. The same 10^{-20} floor as in the reverb keeps denormals out.

The compressor: evening out how loud a part is

Some parts come out unevenly: a bass line where some notes jump out and others get lost. A **compressor** turns a sound down when it gets loud:

- Under its **threshold** it does nothing.
- Over it, it turns the sound down by an amount its **ratio** sets. At 4:1, a sound 8 dB over the threshold comes out only 2 dB over: four decibels in for each one out. At 1:1 it turns nothing down, so the compressor is off. At 20:1 almost nothing gets over the threshold.
- Its **gain** (the "make-up gain") turns the whole part up afterwards, since the loud notes came down. The part is as loud as it was, but steadier.

That rule is plain arithmetic, so it lives in `lineup-core` where a test can check it without any sound:

```
pub fn reduction(&self, level: f32) -> f32 {
    let over = level - self.threshold;
    if over <= 0.0 || self.ratio <= 1.0 {
        0.0
    } else {
        over * (1.0 - 1.0 / self.ratio)
    }
}
```

At 4:1, 8 dB over the threshold is turned down by $8 \times (1 - \frac{1}{4}) = 6$ dB, and comes out 2 dB over.

A compressor does not turn down each wave. It follows how loud the sound has been over the last few milliseconds: it turns down fast once a sound goes over the threshold (its **attack**, 10 ms) and back up more slowly once it is under again (its **release**, 150 ms). If it followed every wave, it would bend the wave's shape and add a buzz. The `Follower` does this:

```
let peak = sample.abs();
let speed = if peak > self.level {
    self.attack
} else {
    self.release
};
self.level += (peak - self.level) * speed;
let level = (20.0 * self.level.log10()).max(FLOOR);
let decibels = compressor.gain - compressor.reduction(level);
10f32.powf(decibels / 20.0)
```

`attack` and `release` are worked out once from their times, as $1 - e^{(-1 / (\text{seconds} \times \text{sample rate}))}$: with that number, the level gets about two thirds of the way to a new loudness in the given time. $20 \times \log_{10}$ turns the level into decibels, `reduction` says how much to turn down, and $10^{(\text{dB} / 20)}$ turns the answer back into a number to multiply the sample by. `.max(FLOOR)` keeps silence, whose logarithm is minus infinity, at -100 dB.

The delay: echoes in time with the song

A **delay** plays the sound again a little later, and again, each echo quieter than the one before. Its **time** is a note value of the song, counted in steps (sixteenths):

```
pub const TIMES: [(u8, &'static str); 6] = [
    (1, "1/16"),
    (2, "1/8"),
    (3, "dotted 1/8"),
    (4, "1/4"),
    (6, "dotted 1/4"),
    (8, "1/2"),
];
```

So the echoes fall on the song's rhythm, and follow the tempo when it changes. The **dotted eighth** is the favourite: its echoes fall three sixteenths after each note, between the beats, and a guitar or a voice playing eighth notes sounds as if it played sixteenths.

```
pub fn seconds(&self, tempo: Tempo) -> f32 {
    let step = tempo.seconds_per_beat() / crate::STEPS_PER_BEAT as f32;
    (f32::from(self.steps) * step).min(Delay::MOST_SECONDS)
}
```

A dotted eighth at 120 BPM is $3 \times 0.125 = 0.375$ s. An echo comes at most 2 seconds after the sound: a half note at 40 BPM would be 3 seconds, so at such slow tempos the longest times stop at 2.

- **Feedback** says how much of each echo comes back as the next one. At 50%, each echo is half as loud as the one before; at 0% there is only one. It stops at 90%, so the echoes always die away.
- **Mix** says how loud the echoes are next to the sound itself. 0% turns the delay off; 100% makes the first echo as loud as the note.

The delay remembers the sound it plays again in a **ring buffer**: a fixed array written round and round, where the sample `delay_frames` back is always behind the one being written:

```
let frames = line.len();
let read = (self.written + frames - self.delay_frames) % frames;
let echo = line.0[read];
line.0[self.written] = sample + echo * self.settings.delay.feedback;
self.written = (self.written + 1) % frames;
sample += echo * mix;
```

What goes into the line is the sound plus part of the echo; that part comes back once more `delay_frames` later, which is the feedback. What comes out is the sound plus the echo at the mix. Adding `frames` before the `%` keeps the subtraction from going below zero, which a `usize` cannot hold.

Memory made on the other side

Two seconds of sound is 96,000 samples at 48 kHz, a big allocation, and the audio thread never allocates. So the window's thread makes the `DelayLine` and sends it with the settings:

```
SetEffects {  
    channel: usize,  
    effects: lineup_core::Effects,  
    line: Option<DelayLine>,  
},
```

It sends a line every time it sends settings with the delay on, because only the audio thread knows whether that channel already has one. `ChannelEffects::set` keeps the first line it gets and returns any other, which goes back on the trash queue to be freed on the window's side:

```
let spare = match self.channels.get_mut(channel) {  
    Some(state) => state.effects.set(effects, line, tempo),  
    None => line,  
};  
if let Some(spare) = spare {  
    self.throw_away(Garbage::DelayLine(spare));  
}
```

A channel keeps its line once it has one. When its delay is turned off and on again, `set` fills the line with silence first, or the echoes of the last time would come back. When the tempo changes, the engine calls `set_tempo` on every channel's effects, so the echoes move with it.

`process` first checks whether any effect is on. If none is, it returns the sample untouched: a channel without effects sounds exactly as it would without the code, and costs nothing.

Level meters: from the audio thread to the screen

A meter shows how loud a channel is **now**. The audio thread knows, since it makes the sound; the window's thread draws the meter. They run at different speeds (the audio thread fills a block of a few hundred samples many times a second, the window draws about 60 frames a second), and the audio thread must never wait for the window, so a `Mutex` is out.

Peaks in atomics

The audio thread keeps the loudest sample of each channel over a block (`channel.peak` above). At the end of the block it raises a shared peak to it, if it is louder. Once a frame, the meter takes the shared peak and sets it back to silence. So no peak is missed, however blocks and frames line up, and neither side waits:

```
/// The peaks since the UI last took them, as the bits of `f32`s.
///
/// For numbers that are not negative, comparing their bits compares the
/// numbers, so `fetch_max` on the bits keeps the louder peak.
pub struct Levels {
    channels: [AtomicU32; MAX_CHANNELS],
    master: [AtomicU32; 2],
}
```

```
fn raise(slot: &AtomicU32, peak: f32) {
    // `peak > 0.0` is false for NaN too, which must not get in.
    if peak > 0.0 {
        slot.fetch_max(peak.to_bits(), Ordering::Relaxed);
    }
}

fn take(slot: &AtomicU32) -> f32 {
    f32::from_bits(slot.swap(0, Ordering::Relaxed))
}
```

Rust's standard library has no `AtomicF32`. But an `f32` is 32 bits, and `to_bits` and `from_bits` turn it into a `u32` and back without changing a bit. Better still, for numbers that are **not negative**, the order of the bits as unsigned integers is the order of the numbers: the exponent sits in the high bits and the mantissa below it. So `fetch_max` on the bits, an atomic “keep the larger”, keeps the louder peak. Peaks are absolute values, never negative, and the `if` keeps NaN out (its bits would compare as larger than any number).

`swap(0, ...)` reads the value and puts 0 (the bits of 0.0) in its place in one step, so a peak raised between a read and a reset can't be lost. `Ordering::Relaxed` is enough because each peak stands alone: nothing else in memory has to be seen in a certain order along with it.

`std::array::from_fn(|_| AtomicU32::new(0))` builds the arrays, since an `AtomicU32` is not `Copy` and `[AtomicU32::new(0); N]` would not compile.

Decibels, and a meter that falls slowly

The ear hears loudness by ratios, not by differences, so the meter is drawn in **decibels**, a scale of ratios:

```
decibels = 20 × log10(level)      1.0 → 0 dB (full scale)
                                   0.5 → -6 dB
                                   0.25 → -12 dB
                                   0.004 → -48 dB (the meter's floor)
```

Each 6 dB down halves the level, wherever on the scale. A meter is lit from -48 dB at the left to 0 dB at the right: green, amber from -6 dB, red at full scale and above. The master's meters measure after the master volume and before `tanh`, so red on the master means the rounding-off is at work. Hovering a meter says its level in decibels.

A meter that followed every peak exactly would flicker too fast to read. So it **jumps up** to a new peak at once and **falls** 24 dB a second:

```
pub fn update(&mut self, levels: &Levels, channels: usize, seconds: f32) {
    let fall = gain(-FALL_DB_PER_SECOND * seconds);
    self.channels.resize(channels, 0.0);
    for (index, shown) in self.channels.iter_mut().enumerate() {
        *shown = levels.take_channel(index).max(*shown * fall);
    }
    for (shown, peak) in self.master.iter_mut().zip(levels.take_master())
    {
        *shown = peak.max(*shown * fall);
    }
}
```

`seconds` is the time since the last frame. Falling 24 dB a second means multiplying by $10^{(-24 \times \text{seconds} / 20)}$ each frame, whatever the frame rate. Half a second after a peak, the meter shows it 12 dB down, at about a quarter.

egui only draws a frame when something happens. While a meter is still falling, the mixer asks for the next frame at once (`request_repaint`); when every meter is dark, it looks again every tenth of a second (`request_repaint_after`), which is enough to catch the notes of a MIDI keyboard played while the song is stopped.

Score order: the drums together

The channel rack keeps channels in the order they were added. A style that brings in a rim shot, a bass, a cowbell and a conga adds them at the end, so a list in rack order scatters the drums around the piano and the bass. A conductor's score never does

that: every instrument has its place, the same in every score, so a reader finds the flutes at the top and the double basses at the bottom without searching. The mixer lists channels in that order, in sections with a heading each:

Section	Instruments, top to bottom
Woodwinds	piccolo, flutes, oboes, English horn, clarinets, saxophones, bassoons
Brass	horns, trumpets, trombones, tuba
Percussion	timpani; glockenspiel, xylophone, vibraphone, marimba, tubular bells; the rest
Drums	hand percussion, then the drum kit
Harp and guitars	harp, guitars, banjo, koto, bass guitar
Keyboards	celesta, piano, harpsichord, organ, accordion and bandoneón
Synths	synth leads, pads and effects; a Lineup Osc with no instrument
Voices	a solo voice, then the choir
Strings	string sections, violins, violas, cellos, double basses

The horns come above the trumpets although they sound lower, because they blend with the woodwinds and were long written next to them. The strings come last, so the double basses, the lowest part of the orchestra, end the page; a choir goes just above them, as in a requiem or an oratorio.

The drums follow the **drum staff**, where each drum has its own line or space: the cymbals on and above the top line (the crash highest, then the hi-hat, then the ride), the high tom in the top space, the snare in the third space, the floor tom below it and the kick in the bottom space. So the **Drums** section reads Crash, Hat, Open hat, Ride, High tom, Snare, Rim (a cross-stick, written where the snare is), Low tom, Kick. Hand percussion (cowbell, shaker, clap, conga) usually has staves of its own above the kit, so it comes first.

Only the mixer changes its order. The channel rack, the piano roll and the saved file keep the channels as they were added, and nothing sounds different.

Where a channel goes

`score_order.rs` is in `lineup-core`, with no UI code, so it is tested with plain `cargo test`. A `Section` is an enum in score order:

```
#[derive(Debug, Clone, Copy, PartialEq, Eq, PartialOrd, Ord, Hash)]
pub enum Section {
    Woodwinds,
    Brass,
    Percussion,
    Drums,
    HarpAndGuitars,
    Keyboards,
    Synths,
    Voices,
    Strings,
}
```

Deriving `PartialOrd` and `Ord` on an enum orders its variants as they are written: `Section::Woodwinds < Section::Brass`. The order of the score is the order of the source code.

The mixer does not look at a channel's name to place it, but at its instrument, the General MIDI program ([SoundFonts and General MIDI](#)). A channel called "Lead" that plays a flute goes with the woodwinds. The program numbers mostly come in families already, but not always the way a score wants: General MIDI puts the timpani with the strings and the celesta with the mallets, and the score puts the timpani first in the percussion and the celesta with the keyboards. So each section lists its programs, top of the score first, and `place` finds a channel's section and its rank in it:

```
pub fn place(channel: &Channel) -> (Section, usize) {
    match &channel.kind {
        ChannelKind::Sampler(SampleSource::Builtin(drum)) => {
            let rank = DRUMS.iter().position(|d| d ==
drum).unwrap_or(DRUMS.len());
            (Section::Drums, rank)
        }
        // A sound file usually takes a drum's row in the rack.
        ChannelKind::Sampler(SampleSource::File(_)) => (Section::Drums,
DRUMS.len() + 1),
        ChannelKind::MidiOut {
            channel: DRUM_CHANNEL,
        } => (Section::Drums, DRUMS.len()),
        ChannelKind::Synth | ChannelKind::MidiOut { .. } => match
channel.program {
            Some(program) => program_place(program),
            // The Lineup Osc, or an outside synth with no instrument
            named.
            None => (Section::Synths, usize::MAX),
        },
    }
}
```

`ChannelKind::MidiOut { channel: DRUM_CHANNEL }` matches only MIDI Out channels whose field equals the constant `DRUM_CHANNEL` (9, shown as 10): a `const` can stand in a pattern like a literal number. The next arm, `MidiOut { .. }`, catches every other MIDI channel; `match` tries the arms from the top, so the drum channel never gets that far.

```
fn program_place(program: u8) -> (Section, usize) {
    Section::ALL
        .into_iter()
        .find_map(|section| {
            let rank = section.programs().iter().position(|&p| p ==
program)?;
            Some((section, rank))
        })
        .unwrap_or((Section::Synths, usize::MAX))
}
```

`find_map` runs the closure on each section and stops at the first one that returns `Some`. Inside, the `?` after `position` returns `None` from the closure when the program is not in that section, which tells `find_map` to try the next. A test checks that each of the 128 programs is in exactly one list, and another that each built-in drum has one place in the kit.

Sorting and grouping

```
pub fn score_order(channels: &[Channel]) -> Vec<(Section, Vec<usize>>> {
    let mut order: Vec<usize> = (0..channels.len()).collect();
    order.sort_by_key(|&index| place(&channels[index]));
    let mut sections: Vec<(Section, Vec<usize>>> = Vec::new();
    for index in order {
        let section = place(&channels[index]).0;
        match sections.last_mut() {
            Some((last, indices)) if *last == section =>
indices.push(index),
            _ => sections.push((section, vec![index])),
        }
    }
    sections
}
```

It sorts the channels' **numbers**, not the channels: each row of the mixer must know which channel it is, to change that channel's volume and to solo the right one. The key is the pair `(Section, usize)`, and tuples compare field by field, so the section decides first and the rank breaks the tie. `sort_by_key` is a *stable* sort: channels with

the same key, such as “Lead”, “Voice 2” and “Voice 3” all playing violins, keep the rack’s order.

The loop then walks the sorted list and starts a new group whenever the section changes. `last_mut` gives a mutable reference to the last group, if there is one; the `if` after the pattern (a *match guard*) only takes the first arm when that group is the same section.

The window

`mixer_window` draws from copies: `mix` is a copy of the project’s `Mixer`, and `levels` a copy of its channels. The sliders change the copies, and when the window is done the app compares them with the project and makes one `edit` for whatever changed, so a drag is one step to undo ([Undo and redo](#)). The master volume goes through `set_master_volume` instead, which reopens the last step if it was also the master volume, so nudging it several times is one step ([Several clicks, one step](#)).

```
for (section, indices) in score_order(&levels) {

ui.label(egui::RichText::new(tr(section.name())).color(theme::ACCENT));
ui.end_row();
for index in indices {
    let channel = &mut levels[index];
    // name, instrument, meter, volume, pan, send, effects, mute ...
    if ui
        .selectable_label(mix.solo == Some(index), tr("S"))
        // ...
        .clicked()
    {
        mix.solo = if mix.solo == Some(index) {
            None
        } else {
            Some(index)
        };
    }
    ui.end_row();
}
}
```

The **Instrument** column is a menu for a synth or MIDI Out channel, with the same choices as the rack’s: the rack and the mixer both call `channel_rack::instrument_choices`. A built-in drum has no instrument to choose, but its menu can send it over MIDI ([MIDI](#)); a sample file’s cell is plain text. The menu is drawn inside closures that already borrow parts of the app, so it cannot change the

project while it is drawn. It pushes what you picked into a list, and the app applies it when the window is done:

```
for action in picked {  
    self.apply(action);  
}
```

It is the pattern the channel rack and the chord sheet use too: the UI code says **what** you did, as a value (`RackAction::SetProgram { channel, program }`), and one place in the app does it.

The **Effects** button says which effects are on (“EQ, delay”), or “none”, and opens a menu with all three. That menu is made with

`PopupCloseBehavior::CloseOnClickOutside`, so it stays open while you drag its sliders, until you click elsewhere. The equalizer’s sliders go back to 0 dB on a double-click, as the pan goes back to the middle.

The window fits its columns, and is fitted again when the channels or their instruments change, since a shorter instrument name can make a column narrower ([Windows the size of what they show](#)).

Testing it

- `reverb.rs`: `bypass_preserves_samples_exactly` (off, the samples come out bit for bit); `impulse_has_a_stereo_tail_that_decays_and_can_be_cleared` at 22.05, 48 and 96 kHz (silence during the pre-delay, a tail that differs left and right, less than a thousandth of the first half second’s energy after one second, and silence after `clear`); `a_tail_ends_in_zeros_not_denormals`.
- `effects.rs` in `lineup-audio` plays sine waves through the effects and measures what comes out: Low at +6 dB turns 50 Hz up by 6 dB and leaves 5 kHz alone; a compressor at -20 dB and 4:1 turns a full-scale wave, 20 dB over, down by 15 dB, and lets a quiet one through; an eighth-note delay at 120 BPM with 50% feedback echoes a click after 0.25 s at full level, then half, then a quarter; a line is kept once and a second one given back.
- `effects.rs` in `lineup-core`: the reduction arithmetic, the delay’s time at 120 BPM and its 2-second cap, settings out of range, the button’s summary (“EQ, comp”), and that plain effects save as `()`.
- `render.rs` plays a click through the whole engine:
`a_panned_channel_plays_more_on_its_side` at five pans against the table above; `a_channel_sent_nowhere_plays_dry_in_a_room` (a send of 0 is as loud as with the reverb off and leaves no tail);

`a_channel_with_a_delay_echoes_in_time_with_the_song;`

`the_meters_see_each_channel_and_the_master` (a click at volume 0.5, panned left, master at 0.8: the channel's meter reads 0.25, the master's left 0.2 and its right 0).

- `levels.rs` and `meters.rs`: the loudest peak is kept until taken and NaN never gets in; a meter jumps up, falls 12 dB in half a second, and goes dark below the floor.
- `score_order.rs`: every program and every drum has exactly one place; drums added after the band join the other drums; an orchestra reads from the flutes down to the basses; the MIDI drum channel and sound files count as drums; channels in the same place keep the rack's order.
- In the app: `the_mixer_pans_a_channel_and_sets_its_reverb_send,`
`the_mixer_sets_a_channels_effects` (opens the menu, picks "dotted 1/4", then "No effects"), `a_click_on_an_instrument_in_the_mixer_changes_it` (and one Undo brings the piano back),
`the_mixer_keeps_the_drums_together_and_each_one_still_solos_alone,` and two exports: `an_exported_song_keeps_each_channel_where_the_mixer_put_it` and `an_exported_song_plays_each_channels_effects.`
- In `file.rs`: `pan_and_reverb_sends_need_format_twenty_six` and `effects_need_format_twenty_eight,` each with a hand-edited file that must be refused.

MIDI: messages, devices, recording and files

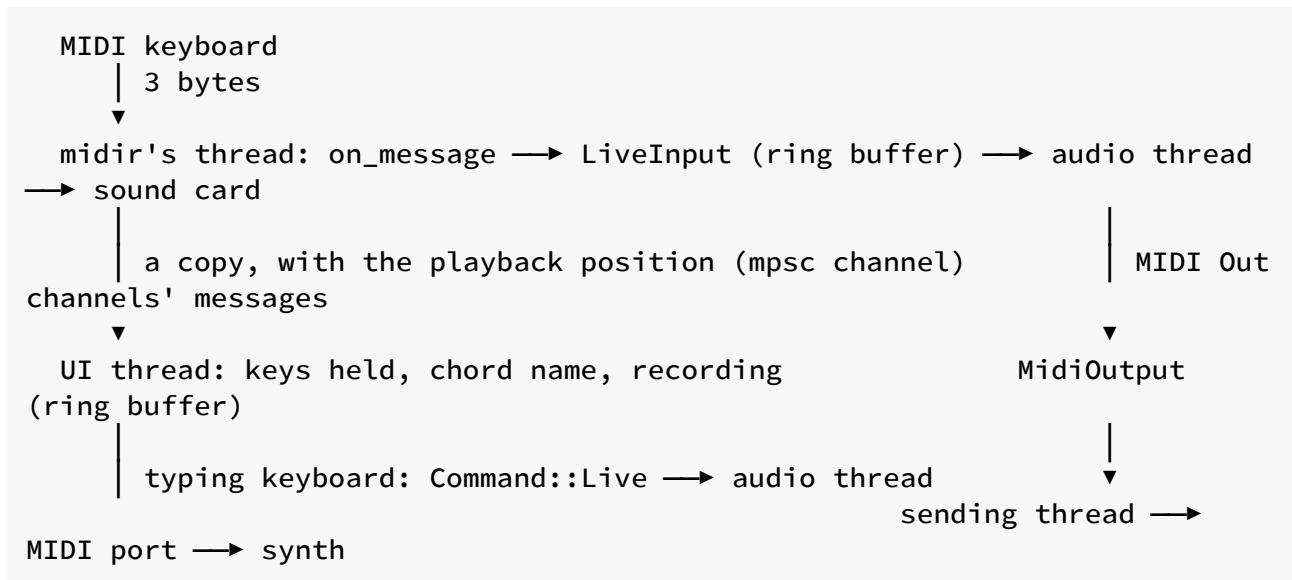
Files: `crates/lineup-core/src/midi.rs`, `crates/lineup-core/src/midi_file.rs`, `crates/lineup-core/src/midi_file/read.rs`, `crates/lineup-core/src/midi_file/import.rs`, `crates/lineup-core/src/record.rs`, `crates/lineup-audio/src/engine.rs`, `crates/lineup-audio/src/instrument.rs`, `crates/lineup-app/src/midi.rs`, `crates/lineup-app/src/app/live.rs` and `crates/lineup-app/src/channel Rack.rs`

MIDI is how keyboards, synths and music programs talk to each other. It has been the same since 1983: a few bytes that say “this key went down, this hard” or “this key came up”. Ferrous Lineup speaks it both ways:

- the keys of a **MIDI keyboard**, or of the computer’s own keyboard laid out as a piano, play the *selected channel*, the one the piano roll shows, and the status bar names the chord they hold;
- **recording** turns the keys played while a pattern loops into notes of that pattern;
- **MIDI Out** sends channels to an external synth instead of the speakers: every channel at once, or the ones you choose;
- **MIDI files** (`.mid`) are written from a song or a pattern, and read back into a new project.

How to use all this is in the manual’s [MIDI keyboards and synths](#), [Opening MIDI files and recordings](#) and [Exporting](#). This chapter follows a key on its way: the bytes, the threads they cross, the note they become, and the file that keeps them.

Four threads share the work. A key from a MIDI keyboard takes this path:



Four ideas carry the design:

1. **Keys skip the UI.** The UI draws once per frame, and a key could wait a whole frame there before anything happens. So a key goes from the MIDI thread straight to the audio thread, and sounds within one audio block, a few milliseconds. The UI gets a copy, to show and record it.
2. **The audio thread still never waits.** Sending a MIDI message is a system call, which may take any time. So the audio thread leaves messages in a queue, and a thread of their own sends them.
3. **One clock for everything.** The audio thread publishes where it is in the pattern, to a fraction of a step. Each key is stamped with it as it arrives, and recording only has to round.
4. **A file is read as data that can be wrong.** A MIDI file from the internet may be cut short or damaged. Every read can fail without a panic, and the import keeps what it could read and says so.

Three bytes per key

A MIDI keyboard says almost everything in messages of three bytes. The first one, the *status byte*, says what kind of message it is and on which of 16 *channels*. The other two are *data bytes*. For a key going down:

```

1001 0000   0011 1100   0110 0100
|           |           |
|           |           |----- velocity 100: how hard the key was struck
|           |           |----- note 60: middle C (C4)
|           |           |----- channel 0 (people call it channel 1)
|----- 1001 = note-on

```

A status byte always has its top bit set (128 and up), and a data byte never does (0 to 127). That is why a MIDI note number and a velocity are never more than 127, and why `Note` uses the same numbers: C4 is 60 in Ferrous Lineup because it is 60 in MIDI.

There are many kinds of message: knobs and pedals (“control change”), pitch bends, clock ticks for keeping machines in time. The app needs three:

```
#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash)]
pub enum MidiMessage {
    NoteOn {
        channel: u8,
        pitch: Note,
        /// How hard the key was struck, 1 to 127.
        velocity: u8,
    },
    NoteOff {
        channel: u8,
        pitch: Note,
    },
    /// From now on, `channel` plays General MIDI program `program`, 0 to
    /// 127. Only sent, never read: a keyboard choosing its own sound does
    /// not change the channel it plays.
    ProgramChange {
        channel: u8,
        program: u8,
    },
}

/// The kinds of message, as the top four bits of the status byte.
const NOTE_OFF: u8 = 0x80;
const NOTE_ON: u8 = 0x90;
const CONTROL_CHANGE: u8 = 0xB0;
const PROGRAM_CHANGE: u8 = 0xC0;
```

The status byte holds two things at once, so the code splits it with bit operations, the ones `pitch-class sets` use too: `status & 0xF0` keeps the top four bits, the kind, and `status & 0x0F` keeps the bottom four, the channel.

Reading a message: slice patterns

```
pub fn parse(bytes: &[u8]) -> Option<MidiMessage> {
    let [status, pitch, velocity, ..] = *bytes else {
        return None;
    };
    // Data bytes have the top bit clear; anything else is malformed.
    if pitch > 127 || velocity > 127 {
        return None;
    }
    let channel = status & 0x0F;
    let pitch = Note::new(pitch);
    match status & 0xF0 {
        NOTE_ON if velocity > 0 => Some(MidiMessage::NoteOn {
            channel,
            pitch,
            velocity,
        }),
        NOTE_ON | NOTE_OFF => Some(MidiMessage::NoteOff { channel, pitch
    })),
        _ => None,
    }
}
```

`let [status, pitch, velocity, ..] = *bytes else` is a **slice pattern**. A slice can have any length, so a pattern for it describes a *shape*: “at least three items; call the first three `status`, `pitch` and `velocity`, and `..` for whatever follows”. If the slice is shorter, the pattern does not match and the `else` block returns. Without slice patterns this would be a length check and three indexings, each of which could panic if the check were wrong. The `*` matches the slice itself, not the reference to it; it is allowed because the parts taken out are `u8`s, which are `Copy`.

The `match` uses two more things. `NOTE_ON` and `NOTE_OFF` are constants, and a constant in a pattern means “equal to this value”, not “a new variable”. And `NOTE_ON if velocity > 0` has a **match guard**: the arm only matches when the condition holds too. A note-on with velocity 0 falls through to the next arm and becomes a note-off. Many keyboards really send that instead of a note-off: a stream of messages of the same kind may leave out the repeated status byte (“running status”), which saved time on the slow cables of 1983.

`parse` reads only notes. Program changes are only ever sent, and every other message, from a sustain pedal to a clock tick, gives `None`.

Writing one: two or three bytes

A note is three bytes, a program change two. `bytes` returns a small value that holds three and says how many count:

```
/// The bytes of one message: two or three of them, as a slice.
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub struct MidiBytes {
    bytes: [u8; 3],
    len: usize,
}

impl std::ops::Deref for MidiBytes {
    type Target = [u8];

    fn deref(&self) -> &[u8] {
        &self.bytes[..self.len]
    }
}
```

Deref is the trait behind the `*` operator. Implementing it says “a `MidiBytes` can be used as a `[u8]`”: `*bytes` gives the slice, and so does passing `&bytes` where a `&[u8]` is wanted, because Rust follows **Deref** by itself there (*deref coercion*). So the sending code can pass `&message.bytes()` to a port, a test can compare `*off.bytes()` with `[0x80, 60, 64]`, and nothing is allocated: the three bytes live in the value, on the stack.

```

pub fn bytes(self) -> MidiBytes {
    let (bytes, len) = match self {
        MidiMessage::NoteOn {
            channel,
            pitch,
            velocity,
        } => (
            [
                NOTE_ON | (channel & 0x0F),
                pitch.midi(),
                velocity.clamp(1, 127),
            ],
            3,
        ),
        // Velocity 64 is the usual "no particular speed" for a release.
        MidiMessage::NoteOff { channel, pitch } => {
            ([NOTE_OFF | (channel & 0x0F), pitch.midi(), 64], 3)
        }
        MidiMessage::ProgramChange { channel, program } => {
            ([PROGRAM_CHANGE | (channel & 0x0F), program & 0x7F, 0], 2)
        }
    };
    MidiBytes { bytes, len }
}

```

The `clamp(1, 127)` matters. A note-on with velocity 0 would be read as a note-off by the synth, so a note played “as softly as possible” would not be played at all. `& 0x0F` and `& 0x7F` keep a wrong channel or program number from spilling into the bits of another field.

One message is written and never read: control change 123, “All Notes Off”. It tells a synth to stop every note on a channel:

```

pub fn all_notes_off(channel: u8) -> [u8; 3] {
    [CONTROL_CHANGE | (channel & 0x0F), 123, 0]
}

```

The app sends it on all 16 channels whenever it lets go of a synth, because a note-off lost on the way would leave a note sounding for ever: when another output port is picked, and when the audio thread that owed the note-offs goes away ([Two threads](#)).

A key’s way in

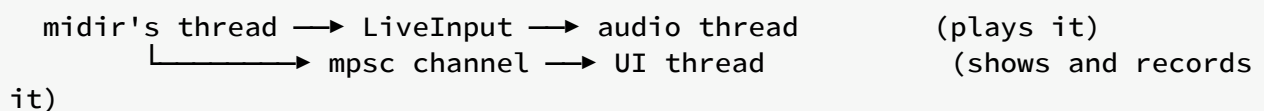
Most notes come from the sequencer, which lives on the audio thread and knows each note’s length before it starts. A key on a keyboard is different in two ways: it

comes from outside, on a thread Ferrous Lineup does not own, and nobody knows how long it will be held.

Why keys skip the UI

The obvious path for a key would be the one every other change takes: the UI hears about it, and sends a `Command` to the audio thread. But the UI only runs once per frame. At 60 frames a second that is up to 17 ms of waiting, and on a slow computer much more, before the audio thread even hears of the key. A pianist feels that.

The audio thread runs far more often: once per *block*, every few milliseconds. So a key from a MIDI keyboard goes straight there, and the UI gets a copy afterwards, only to show it and record it:



The typing keyboard cannot do this: egui reads the keyboard, and egui runs on the UI thread. So typed keys take the slow way, as a `Command::Live(MidiMessage)`. On the audio thread both end up in the same method, `play_live`.

One producer per queue: LiveInput

The command queue is a ring buffer from the `rtrb` crate ([Two threads](#)), and its `Producer` belongs to the `AudioEngine`, on the UI thread. `rtrb` is *single producer, single consumer*: one thread pushes, one thread pops, and that is what lets it work without locks. `midir's thread` is a second thread that wants to push, so it needs a ring buffer of its own:

```

pub struct LiveInput {
    notes: rtrb::Producer<MidiMessage>,
    position: Arc<AtomicU64>,
}

impl LiveInput {
    /// Plays (or lets go of) a key on the live channel. Returns `false`
    /// if the queue was full and the key was lost.
    pub fn send(&mut self, message: MidiMessage) -> bool {
        self.notes.push(message).is_ok()
    }

    /// Where playback is, in steps (see [Sequencer::position]), or
    /// `None` when stopped. Recording uses it to place the key.
    pub fn position(&self) -> Option<f64> {
        load_position(&self.position)
    }
}

```

Because there can be only one producer, this cannot be a method on `AudioEngine` that anybody may call. It is a *value*, and whoever owns it is the one producer. The engine creates it when it starts and keeps it in an `Option` until someone asks for it:

```

pub fn take_live_input(&mut self) -> Option<LiveInput> {
    self.live_input.take()
}

```

`Option::take` moves the value out and leaves `None` behind, so the second call gets nothing. The type system makes the rule “one producer” impossible to break: `LiveInput` does not implement `Clone`, so it cannot be copied, only moved. `MidiOutput`, the way out for MIDI Out channels, works the same way from the other end: it holds the queue’s `Consumer`, and `take_midi_output` hands it over once.

On the audio thread, the consumer is an `Option` too, because the audio thread that [exports a song](#) has no keyboard at all:

```

while let Some(message) = self.live_input.as_mut().and_then(|keys|
    keys.pop().ok()) {
    self.play_live(message);
}

```

That line packs three steps. `self.live_input` is an `Option<Consumer>`, and calling `pop` needs a `&mut Consumer`. `as_mut()` turns `&mut Option<Consumer>` into `Option<&mut Consumer>`, borrowing what is inside instead of moving it out. `and_then` runs the closure only if there is a consumer, and the closure’s own `Option` (from

`pop().ok() : Some(key)` , or `None` when the queue is empty) becomes the result. So the loop goes on while there is a consumer *and* it has a key.

Where in the pattern: the position

Recording needs to know *when* each key came, as a place in the pattern: “halfway through step 4”. The sequencer can tell, since it counts down the frames until the next step ([The sequencer](#)):

```
pub fn position(&self) -> f64 {
    if self.frames_until_next <= 0.0 {
        return self.next_step as f64;
    }
    let current = (self.next_step + self.length - 1) % self.length;
    let passed = 1.0 - self.frames_until_next / self.frames_per_step;
    current as f64 + passed.clamp(0.0, 1.0)
}
```

`next_step` is the step about to fire, so the one playing is the step before it. `next_step - 1` would go below zero at step 0, which a `usize` cannot do (it would panic in a debug build). Adding `length` first and taking the remainder goes back one step *around the loop*: from step 0 to the last step. `passed` is how much of that step is over, from 0 to 1.

The audio thread stores it after every block, as the bits of the `f64` in an `AtomicU64` , since there is no atomic float ([Two threads](#)). “Stopped” is `NO_POSITION = u64::MAX` ; as a float, those bits are a NaN (“not a number”), which no position ever is. `LiveInput` holds a clone of the same `Arc` , so `midir`’s thread can read the position at the very moment a key arrives. It is up to one block old then, about a hundredth of a second; a step at 120 BPM lasts an eighth of a second, and recording rounds to the nearest step anyway.

A key on the audio thread

A note of the song gets a gate, the number of frames it is held. A key has no known length, so it gets `HELD` , a gate that never runs out: only `note_off` ends it ([Instruments](#)). `play_live` starts and ends them:

```

fn play_live(&mut self, message: MidiMessage) {
    match message {
        MidiMessage::NoteOn {
            pitch, velocity, ..
        } => {
            // A key that is already down (on both keyboards at once,
            // say) comes up first.
            self.release_live(pitch);
            if let Some(channel) = self.live_channel
                && let Some(state) = self.channels.get_mut(channel)
            {
                state
                    .instrument
                    .note_on(pitch, velocity, HELD, self.sample_rate);
                self.live_keys[usize::from(pitch.midi())] = Some(channel);
            }
        }
        MidiMessage::NoteOff { pitch, .. } => self.release_live(pitch),
        MidiMessage::ProgramChange { .. } => {}
    }
}

```

Keys play the *live channel*, which the UI sets with `Command::SetLiveChannel` whenever the selected channel changes. But the selection can change while a key is down. If the key then came up on the new channel, the note on the old one would play for ever. So the audio thread remembers, for each of the 128 notes, which channel its key went down on:

```

live_keys: [Option<usize>; 128],

fn release_live(&mut self, pitch: Note) {
    if let Some(channel) =
self.live_keys[usize::from(pitch.midi())].take()
    && let Some(state) = self.channels.get_mut(channel)
    {
        state.instrument.note_off(pitch);
    }
}

```

An array of 128 `Option`s is a map from note to channel that never allocates, and `take()` reads an entry and empties it in one go. The `if let ... && let ...` is a **let chain**: both patterns must match for the block to run.

Channels are known by their position in the list, and inserting or removing a channel moves the ones after it. A remembered “channel 2” could then be a different channel, or none. So before the list changes, `release_live_keys` lets go of every key that is down. The test `removing_channels_lets_go_of_their_keys` removes channel 0 while a key is down on channel 1, which then *becomes* channel 0.

Live notes play like a preview: on a muted channel too, and whether or not the sequencer is playing. Muting stops the song's notes on a channel, not the key you are holding (`muting_a_channel_stops_its_notes_but_not_keys_held_down`). Stopping playback lets go of everything, keys included.

midir: ports and connections

`midir` is the crate that talks to each system's MIDI layer: ALSA on Linux, Core MIDI on macOS, Windows Multimedia on Windows. It is built the way `cpal` is: list the ports, open a connection to one, and the connection stays open for as long as the value is kept. Ferrous Lineup wraps it in a `Midi` struct, in `lineup-app/src/midi.rs`.

Whether there is MIDI at all is found out by trying: creating a `MidiInput` fails on a system without a MIDI layer, and the error is kept to show in the menu. On Linux that is a computer without the ALSA *sequencer* (the `snd-seq` kernel module, which makes `/dev/snd/seq`), and the menu's hint says so.

Ports are found again by an id that `midir` gives, a `String` that stays the same while the device is plugged in, because the list of ports changes as devices come and go. The MIDI menu looks for ports each time it opens (`refresh_ports`), and leaves out Ferrous Lineup's own: on Linux, the port the app sends from shows up as an *input* too, and picking it would send every note round in a circle.

```
fn is_own(name: &str) -> bool {
    name.starts_with(&format!("{CLIENT_NAME}:"))
}
```

The colon matters: a synth called "Ferrous Lineupish Synth" is not the app (the test `its_own_ports_are_left_out` checks that).

Connecting consumes, closing gives back

```
let mut midi_in = MidiInput::new(CLIENT_NAME).map_err(|err|
err.to_string());
// Clock ticks, "I am still here" messages and system exclusive
// data are not notes. Leaving them out saves waking up for them.
midi_in.ignore(Ignore::All);
let found = midi_in
    .find_port_by_id(&port.id)
    .ok_or_else(|| crate::i18n::trf("{} is not connected any more", &
[&port.name]))?;
let link = InputLink {
    live: self.live.clone(),
    to_ui: self.to_ui.clone(),
    ctx: self.ctx.clone(),
    snap: Arc::clone(&self.snap),
    sounding: Vec::new(),
};
let connection = midi_in
    .connect(&found, "input", on_message, link)
    .map_err(|err| err.to_string());
```

`connect` takes `self`, not `&self`: the `MidiInput` is consumed and lives inside the connection from then on. `close` gives it back, with the data. This is ownership used as a state machine: a `MidiInput` that is connected cannot be connected twice, because you no longer have it.

`on_message` is a plain function, and `link` is the data it gets each time, as `&mut InputLink`:

```
fn on_message(_microseconds: u64, bytes: &[u8], link: &mut InputLink) {
    let Some(message) = MidiMessage::parse(bytes) else {
        return;
    };
    let message = link.snapped(message);
    let mut position = None;
    if let Some(live) = &link.live {
        let mut live = lock(live);
        live.send(message);
        position = live.position();
    }
    // `send` fails only if the UI is gone, and then nobody needs the
    copy.
    let _ = link.to_ui.send(Played { message, position });
    // egui only redraws when something happens. This is something, and
    // `request_repaint` is one of the few egui calls meant for other
    threads.
    link.ctx.request_repaint();
}
```

It runs on midir's thread, so `connect` asks for it and its data to be `Send + 'static`, like the audio callback. The time midir passes in is ignored: the position from the audio thread is the clock that matters.

Why the LiveInput sits behind a Mutex

`InputLink` holds the `LiveInput` as `Arc<Mutex<LiveInput>>`, which looks odd for a value only one thread uses at a time. The reason is midir's errors. If `connect` fails, the error gives back the `MidiInput`, but *not* the data: `link` would be gone, and with it the one producer of the live queue, for good. No keyboard could ever be connected again.

So the `Midi` struct keeps the `LiveInput` itself, behind an `Arc<Mutex<>>`, and each link holds a clone of the `Arc`. A failed connection loses only a clone. Only one connection exists at a time, so the lock is never waited on, and midir's thread is not the audio thread, so it may lock. The same `Arc` lets the app swap in a new engine's `LiveInput` after the sound card was opened again (`reconnect_audio`), without reconnecting the keyboard.

Poisoned locks

```
fn lock<T>(mutex: &Mutex<T>) -> MutexGuard<'_, T> {
    mutex.lock().unwrap_or_else(PoisonError::into_inner)
}
```

If a thread panics while it holds a lock, Rust marks the `Mutex` as **poisoned**, and every later `lock()` returns an `Err`: the data inside may be half changed, and Rust will not hand it over without asking. `PoisonError::into_inner` is the way to say "give it to me anyway". Here it is safe, because a queue end, a connection or a key is never left half changed. The helper is generic over `T`, so it works for every mutex in the file.

A copy for the UI: mpsc

The UI's copy of each key goes through a `std::sync::mpsc` channel ("multiple producer, single consumer"), from the standard library:

```
let (to_ui, from_keyboard) = mpsc::channel();
// ...
pub fn played(&self) -> Vec<Played> {
    self.from_keyboard.try_iter().collect()
}
```

Unlike `rtrb`, it allocates and may lock, which is fine here: neither `midir`'s thread nor the UI thread is real-time. And the `Sender` can be cloned, so each new connection gets its own. `try_iter` gives the messages that are waiting and stops when there are none, without waiting for more: right for a UI that must not stop drawing.

`egui` only draws a frame when something happens: the mouse moves, a key is typed. A key on a MIDI keyboard is not something `egui` sees, so `on_message` calls `ctx.request_repaint()`. `egui::Context` is a handle that can be cloned and sent to other threads, and `request_repaint` is one of the few `egui` calls meant to be used from them.

Snapping keys to the key

The piano roll's **Snap notes you play** moves every key you play to the nearest note of the key the song is in at that point, so a wrong note cannot happen ([Scales, modes and keys](#)). For a MIDI keyboard that has to happen on `midir`'s thread, before the key goes to the audio thread. The UI shares the key to snap to through one more `Arc<Mutex<Option<Scale>>>`, set once per frame, and the link snaps each key with `tools::nearest_in_key` ([Editing tools as plain functions](#)).

The key of the song can change while a key is held: playback crosses a key marker. Then the key-up, snapped to the new key, could name another note, and the note started by the key-down would never end. So the link remembers, for each key that is down, the note it plays:

```
MidiMessage::NoteOff {
    channel,
    pitch: key,
} => {
    let at = self.sounding.iter().position(|&(k, _)| k == key);
    let pitch = at.map_or(key, |at| self.sounding.remove(at).1);
    MidiMessage::NoteOff { channel, pitch }
}
```

`map_or(key, ...)` reads “the note this key started, or the key itself if it started nothing we know of”. The typing keyboard keeps the same pairs for its keys, in `typing_down`.

The typing keyboard

No MIDI keyboard at hand? **MIDI > Typing keyboard to piano** turns the letter keys into one, laid out the way a piano's black keys sit between and above the white ones:

black keys:	S	D		G	H	J		2	3		5	6	7		9	0	
white keys:	Z	X	C	V	B	N	M	Q	W	E	R	T	Y	U	I	O	P
notes:	C4	D4	E4	F4	G4	A4	B4	C5	D5	E5	F5	G5	A5	B5	C6	D6	E6

The layout is a table, `TYPING_KEYS: &[(Key, u8)]`, and `typing_note` looks a key up in it with `find`. Reading the keys happens once per frame. `egui` keeps the keyboard's events of this frame in a list, and the code picks out the ones it wants:

```
egui::Event::Key {
    key,
    physical_key,
    pressed,
    repeat: false,
    modifiers,
} => {
    // The *physical* key, where it is on the keyboard,
    // so the piano keeps its shape on any layout:
    // on a French keyboard, W is where Z is in English.
    let key = physical_key.unwrap_or(*key);
    // Ctrl+Z is undo, not a note. Key releases always
    // count, or a note could stay on.
    let shortcut = modifiers.command || modifiers.alt;
    (!*pressed || !shortcut).then_some((key, *pressed))
}
```

Four details make it behave like a piano:

- **repeat: false** in the pattern. A key held down makes the system repeat it ("aaaaa" in a text field). A field written as a value in a pattern must equal that value, so repeats do not match, and a held key plays once.
- **physical_key**. `key` is the letter the keyboard layout prints on the key; `physical_key` is where the key *is*. Some systems do not report it, so `unwrap_or` falls back to the letter.
- **Shortcuts are not notes**. A key going down with Ctrl (Cmd on a Mac) or Alt held is skipped. A key coming *up* always counts, whatever is held, or a note started before Ctrl was pressed would never end. The app's one-key shortcuts (Space, L, Home, End, Delete...) are not on the piano's keys, so they still work while it is on.
- **Focus**. If the window loses the keyboard (you click another window), or a text field, a menu or a dialog takes it, the keys' "up" events go elsewhere. So all typing keys are let go at once when that happens.

The events are read inside `ctx.input(|input| ...)`, which holds a lock on egui's input while the closure runs. The closure only collects a `Vec` of `(key, pressed)`, and the loop that plays the notes runs after it, with the lock released.

Letting go of all keys uses `std::mem::take`:

```
fn release_typing_keys(&mut self) {
    // `take` leaves an empty list behind, so the loop can call `&mut self` methods.
    for (_, pitch) in std::mem::take(&mut self.live.typing_down) {
        self.play_typed(MidiMessage::NoteOff { channel: 0, pitch });
    }
}
```

Looping over `&self.live.typing_down` would borrow `self` for the whole loop, and `play_typed` needs `&mut self`. Taking the list out first leaves an empty one in its place, and the loop owns the old one, so the two never overlap.

`play_typed` sends the key to the audio thread as `Command::Live`, then hands it to `key_played` with the position, exactly as a key from `midir` arrives. The [harmony keys](#) play their chords the same way. From there on, the app cannot tell the keyboards apart.

The selected channel

Keys need a channel to play, and the piano roll already has one: the channel it shows. Once per frame, `live_input` tells the audio thread if that changed:

```
let channel = self
    .piano_roll
    .selected_channel()
    .filter(|&channel| channel < self.project.channels().len());
if self.live.channel_sent != Some(channel) {
    self.send(Command::SetLiveChannel(channel));
    self.live.channel_sent = Some(channel);
}
```

`channel_sent` is an `Option<Option<usize>>`, and both layers mean something. The outer `None` is “nothing sent yet”, so the first frame, and the first frame after a new project, always sends. `Some(None)` is “sent: no channel”, which the audio thread takes as “let go of every key and ignore new ones”. Folding the two into one `Option` would make one of them impossible to say.

Choosing another channel also ends a take that is being recorded, and lets go of every key that is down, so a key never ends on a channel it did not start on.

Keys held

Every key that is down, from either keyboard, is kept in a list of `Held`, in the order they went down:

```
struct Held {
    pitch: Note,
    velocity: u8,
    /// The channel it plays, and will be recorded into.
    channel: usize,
    /// Where playback was when it went down, if it is being recorded.
    recorded_from: Option<f64>,
}
```

This list is what the rest of the app means by “the keys you hold”. The status bar names their chord (“ Am: A minor, chord vi of C major”), the piano roll lights them up on its keyboard, the harmony keys and the key finder listen to them ([Hearing harmony](#)), and **Accompaniment follows chords** turns them into the band’s chord ([How the band plays a chord sheet](#)). Recording is one more use.

Recording: from a key to a note

The  button next to Play only *arms* recording. Keys are recorded while four things are true at once:

```
pub(super) fn is_recording(&self) -> bool {
    self.live.recording
    && self.playing
    && self.mode == PlayMode::Pattern
    && self.piano_roll.selected_channel().is_some()
}
```

Armed, playing, in pattern mode (a take goes into the pattern that loops), and with a channel to record into. Arming switches to pattern mode for you, so  and then Space is enough.

Remembering where a key went down

When a key goes down, `key_played` adds a `Held`:

```
self.live.held.retain(|held| held.pitch != pitch);
self.live.follow.key_down();
let recorded_from = played.position.filter(|_| self.is_recording());
self.live.held.push(Held {
    pitch,
    velocity,
    channel,
    recorded_from,
});
```

`retain` first removes the same pitch if it was already down (on the other keyboard, say), so each pitch is in the list once.

`Option::filter` keeps a `Some` only if the closure says yes, and turns it into `None` otherwise. So `recorded_from` is the position when the key is being recorded, and `None` when it is just being played. The closure ignores its argument (`|_|`): the question is not about the position, but about whether recording is on.

When the key comes up, its `Held` is taken out of the list, and if it has a start position and there is an end position, it becomes a note:

```
let held = self.live.held.remove(index);
if let (Some(from), Some(until)) = (held.recorded_from, played.position) {
    self.record(held, from, until);
}
```

Matching a tuple of two `Option`s checks both at once: the block runs only if both are `Some`.

Quantizing: recorded_note

Nobody plays exactly on the step. A key that went down at 3.9 and came up at 6.2 was meant as a note from step 4 to step 6. Moving the ends of a note to the grid is called **quantizing**, and it is a plain function in `lineup-core`, with no keyboard and no UI in sight:

```

pub fn recorded_note(pitch: Note, velocity: u8, from: f64, until: f64,
length: usize) -> NoteEvent {
    let length = length.max(1);
    let until = if until < from {
        until + length as f64
    } else {
        until
    };
    let start = from.round();
    let end = until.round().max(start + 1.0);
    // `as usize` turns a float into an integer, dropping the fraction
    // (there is none left after `round`). Negative numbers become 0.
    let start_step = start as usize % length;
    let steps = ((end - start) as usize).min(length - start_step);
    NoteEvent {
        velocity,
        ..NoteEvent::new(pitch, start_step, steps)
    }
}

```

Line by line:

1. `length.max(1)` guards the `%` below: a remainder by zero panics. A pattern is never empty, but a function should not depend on its caller for that.
2. The pattern loops. If the key came up *before* where it went down (down at 14.5, up at 1.5), the loop started again in between, and the end is really 17.5, one loop further on.
3. `round()` moves each end to the nearest step. `f64::round` rounds halves away from zero, so 2.5 becomes 3.
4. `.max(start + 1.0)`: a quick tap would round both ends to the same step, a note of length 0. A note is at least one step long.
5. A key pressed just before the loop starts again (at 15.7 in a 16-step pattern) rounds to 16, which is step 0 of the next time round: `% length` folds it back to 0. That is where the player meant it.
6. A note cannot run past the end of the pattern, so it is cut there with `.min(length - start_step)`.

The last lines use **struct update syntax**: `..NoteEvent::new(...)` means “every field not named, as in this value”, so the note gets its velocity and `nudge` and `trim` of 0 from `new`. A recorded note lands on the grid.

The tests read like those rules:

```

#[test]
fn ends_go_to_the_nearest_step() {
    assert_eq!(recorded(2.0, 4.0), (2, 2));
    assert_eq!(recorded(1.8, 4.3), (2, 2), "a little early, a little
late");
    assert_eq!(recorded(2.4, 4.6), (2, 3));
}

#[test]
fn just_before_the_loop_is_step_zero() {
    assert_eq!(recorded(15.7, 0.4), (0, 1));
    assert_eq!(recorded(15.6, 1.9), (0, 2));
}

#[test]
fn a_note_across_the_loop_end_is_cut_there() {
    assert_eq!(recorded(14.0, 2.0), (14, 2));
}

```

`recorded(from, until)` is a helper in the test module that records a C4 in a 16-step pattern and returns `(start, length)`. Being able to write `15.7` instead of having to play a key at exactly that moment is why the function lives in `lineup-core` and takes plain numbers.

`start as usize` turns an `f64` into a `usize`. For floats, `as` drops the fraction, and it **saturates**: a negative number becomes 0, a number too big becomes `usize::MAX`, and NaN becomes 0. It never panics and never wraps around. That is different from `as` between integers, which keeps only the low bits: `300_i32 as u8` is 44.

The positions say *where in the loop* a key was, not *which time round*. A key held for more than a whole loop looks the same as a short one: down at 2, up at 3 one loop later reads as one step long. And a note across the end of the loop is cut there, instead of going on at step 0. Both are left as [exercises](#).

One undo step per take

`record` adds the note to the selected pattern through `edit_pattern`, like every other change, so it is undoable and the audio thread hears about it ([Telling the audio thread what changed](#)):

```

fn record(&mut self, held: Held, from: f64, until: f64) {
    let Some(length) =
self.project.pattern(self.pattern).map(Pattern::length) else {
        return;
    };
    let note = recorded_note(held.pitch, held.velocity, from, until,
length);
    self.edit_pattern("record notes", |pattern| {
        pattern.add_note(held.channel, note);
    });
}

```

`.map(Pattern::length)` passes a method as the function to `map: Pattern::length` is a function that takes a `&Pattern`, exactly what `map` hands over. `add_note` skips a note with the same pitch and start as one already there, so playing the same thing the next time round does not stack notes on top of each other.

A take can be dozens of notes, and Ctrl+Z should undo all of them, not one at a time. The undo history already keeps a step open while the mouse is held, so a drag is one step ([Undo and redo](#)). Recording joins that rule:

```

pub(super) fn end_frame_for_undo(&mut self, ctx: &egui::Context) {
    let holding = ctx.input(|input| input.pointer.any_down());
    if !holding && !ctx.text_edit_focused() && !self.is_recording() {
        self.close_step();
    } else if self.step_unsynced {
        // ...
    }
}

```

While recording, the step stays open, and every note joins it. The take ends, and the step with it, when playback stops, recording is disarmed, the mode changes or another channel is chosen.

Ending a take: before Stop

When playback stops, some keys may still be down. They are recorded as if they came up right then, by `finish_recording`:

```
pub(super) fn finish_recording(&mut self) {
    let until = self.position();
    for index in 0..self.live.held.len() {
        let held = self.live.held[index];
        if let (Some(from), Some(until)) = (held.recorded_from, until) {
            self.record(held, from, until);
        }
        self.live.held[index].recorded_from = None;
    }
}
```

The keys stay in the list (they are still down, and still sound), but their `recorded_from` is cleared, so their key-up later records nothing. The loop goes by index, not `for held in &self.live.held`, because `record` needs `&mut self` while the list is borrowed. `Held` is `Copy`, so `let held = self.live.held[index];` copies it out, and the borrow ends on that line.

The order in `halt`, which every way of stopping goes through, matters:

```
fn halt(&mut self) {
    if self.playing {
        // Before the audio thread forgets where it was.
        self.finish_recording();
    }
    self.playing = false;
    self.send(Command::Stop);
}
```

`finish_recording` needs the position, and `position()` gives `None` once `self.playing` is false, or once the audio thread has stopped and stored `NO_POSITION`. So the take is finished first, while both still say “playing”, and only then is `Stop` sent.

Latency: what you hear is in the past

When you press a key “on the beat”, you press it on the beat you *heard*. But sound takes time to reach you. The audio thread makes each block ahead of time and hands it to the sound card, which plays it a little later; a sound card usually holds one or two blocks, so with blocks of 512 frames at 48 kHz a sound comes out 11 to 21 ms after the audio thread made it. By the time you hear step 4 begin, the audio thread is already that far into it, and a key pressed exactly then is stamped that late.

At 120 BPM a step lasts 125 ms, so 21 ms is a sixth of a step, and rounding to the nearest step hides it, unless you were already a little late. A MIDI keyboard adds almost nothing on top: its keys are stamped on `midir`’s thread as they arrive. The

typing keyboard adds more: egui reads the keyboard once per frame, so a typed key can wait up to a frame before it is stamped, 17 ms at 60 frames a second, and much more on a computer that draws its frames slowly.

The fix is to subtract the latency from each key's position before rounding. cpal can tell how long it is: each call of the audio callback gets an `OutputCallbackInfo` with two timestamps, when the callback ran and when its sound will be played. That is one of the [exercises](#).

MIDI Out: an instrument with an outbox

A MIDI Out channel plays no sound. Its instrument turns notes into MIDI messages for another synth. It is one more kind of `Instrument`, next to the sampler, the Lineup Osc and the SoundFont player, so everything else (the sequencer, live keys, volume, mute) works for it without changes.

The audio thread must not send the messages itself. Sending goes through the operating system's MIDI layer, a system call that may take any time, and the [golden rule](#) forbids waiting. So `MidiOut` collects its messages in an *outbox*, a `Vec` with room reserved when it is created on the UI thread:

```
fn post(&mut self, message: MidiMessage) {
    let outbox = &mut self.state.outbox;
    if outbox.len() < outbox.capacity() {
        outbox.push(message);
    }
}
```

Checking `len() < capacity()` before pushing means the `Vec` never grows, so it never allocates. The room is `OUTBOX_SIZE`, 256 messages for one block of a few milliseconds; even a sixteen-note chord starting while sixteen others end is only 32. If it were ever full, a message would be lost rather than break the rule.

```
pub struct MidiOut {
    /// The MIDI channel, 0 to 15.
    channel: u8,
    /// The note every note goes out as, for a drum.
    pitch: Option<Note>,
    /// The General MIDI program the synth should play, if any.
    program: Option<u8>,
    /// Whether the program change still has to be sent: before the first
    /// note, and again after every stop, in case the synth's sound was
    /// changed on the synth itself in between.
    program_due: bool,
    /// Frames played so far: the clock that says when a note-off is due.
    now: u64,
    /// Boxed so an `Instrument` stays small, as the voices of the others
    are.
    state: Box<MidiOutState>,
}
```

A synth only ends a note when it is told to, so `MidiOut` must send a note-off when each note's gate runs out. It keeps, for each of the 128 notes, an enum that says whether the note is sounding and until when:

```
enum Sounding {
    Off,
    /// Until this frame, when its note-off is due.
    Until(u64),
    /// Until `note_off`.
    Held,
}
```

`next_sample` makes no sound, but it counts the frames in `now`, which is the clock for `Until`. Starting a note posts a program change first when one is due, then the note-on:

```
let pitch = self.pitch.unwrap_or(pitch);
let slot = usize::from(pitch.midi());
// MIDI has one of each note per channel. If this one is already
// sounding, end it first, or the synth would hear "on, on, off"
// and the new note would stop at the old one's note-off.
if self.state.notes[slot] != Sounding::Off {
    self.post(self.note_off_message(pitch));
}
```

`self.pitch.unwrap_or(pitch)` reads “the drum's note if there is one, otherwise the note itself”, and writing `let pitch = ...` again *shadows* the parameter: from that line on, `pitch` means the new value, and the old one cannot be used by mistake.

`note_off` ends only notes that are `Held`: if the pattern plays a long C4 and you tap C4 on the keyboard, letting go of your key must not end the pattern's note.

Once per block, the audio thread asks every instrument to hand over its messages:

```
pub fn flush(&mut self, mut send: impl FnMut(MidiMessage)) {
    for index in 0..128 {
        if let Sounding::Until(end) = self.state.notes[index]
            && end <= self.now
        {
            self.post(self.note_off_message(Note::new(index as u8)));
            self.state.notes[index] = Sounding::Off;
        }
    }
    // `drain(..)` empties the `Vec` but keeps its memory for next time.
    for message in self.state.outbox.drain(..) {
        send(message);
    }
}
```

Note-offs are checked once per block, not on every frame, so they can be a few milliseconds late, which nobody hears at the end of a note. `drain(..)` takes every item out of the `Vec` in order, and the `Vec` keeps its memory for the next block. `send` is an `impl FnMut`: the instrument does not know or care where the messages go. The audio thread passes a closure that puts them in the `MidiOutput` queue (or, when that is full, a backlog reserved in advance, because a lost note-off leaves a note sounding for ever; see [Two threads](#)). The tests pass one that pushes them into a plain `Vec` to look at.

The state lives in a `Box`. 128 `Sounding`s and 128 `Option<Origin>`s take a few kilobytes, and an enum is as big as its biggest variant: without the box, every `Instrument`, and every `Command` that carries one through the ring buffer, would be that big. An engine test keeps a command at 80 bytes or less, and it would fail without the box.

A SoundFont instrument is a `MidiOut` too, whose messages go to a synthesizer inside the app instead of a port ([SoundFonts and General MIDI](#)).

The sending thread

The messages that MIDI Out channels leave in the `MidiOutput` queue are sent by a thread of their own, named `midi-out`:

```

let sender = move || {
    while !messages.is_abandoned() {
        thread::sleep(SEND_INTERVAL);
        let mut output = lock(&output);
        while let Some(message) = messages.pop() {
            // With no port connected, the messages are dropped. A send
            // that fails (a cable pulled out) loses only that message.
            if let Some(connection) = output.as_mut() {
                let _ = connection.send(&message.bytes());
            }
        }
    }
    // ...
};
// If the thread cannot start, MIDI Out stays silent; nothing else breaks.
thread::Builder::new()
    .name("midi-out".to_string())
    .spawn(sender)
    .ok()

```

It sleeps for a millisecond (`SEND_INTERVAL`), sends whatever is waiting, and starts again. A thread that waits for work would usually sleep until it is woken, with a `Condvar` . But waking a thread is a system call, which the audio thread must not make. Looking a thousand times a second costs almost nothing, and adds at most a millisecond.

The lock is taken after the sleep, not before, so the UI thread never waits for a sleeping thread. The `output` guard is dropped at the end of each turn of the loop, which unlocks it.

`is_abandoned()` is true once the other end of the queue is gone: the audio engine was dropped, as when the app closes or the sound card is opened again. Then the loop ends, the thread sends All Notes Off for the notes the old engine still owed, and finishes. Nothing has to tell it to stop.

`thread::Builder` is the long way to start a thread. It can give the thread a name, which shows in debuggers and in tools like `top -H` . And its `spawn` returns a `Result` where `thread::spawn` would panic if the system refused. `.ok()` turns it into an `Option<JoinHandle>` : if the thread cannot start, MIDI Out stays silent and everything else works, which is better than a crash at start. Without audio there is no sending thread at all, since the notes are made on the audio thread, and the MIDI menu says that nothing can be sent.

Changing ports without hanging notes

The connection to the output port sits behind a `Mutex` because two threads use it: the sending thread sends, and the UI thread replaces it when you pick another port:

```
// Swap the connections while holding the lock, so the sending
// thread never sends half to one and half to the other.
let old = std::mem::replace(&mut *lock(&self.output), new);
if let Some(mut old) = old {
    // Notes that were playing there would hang: tell them all to stop.
    for channel in 0..16 {
        let _ = old.send(&all_notes_off(channel));
    }
    // `close` gives back the `MidiOutput`, which is not needed.
    old.close();
}
```

`&mut *lock(&self.output)` locks the mutex, follows the guard to the `Option` inside `(*)`, and borrows it mutably, all in one expression; the guard is dropped at the end of the statement, so the lock is held just long enough to swap. The sending thread then sends either to the old port or the new one, never half to each.

The input side has its own version of this: when you pick another MIDI keyboard, every key held on the old one is let go (`release_all_keys`), because their key-ups would never arrive.

Sending channels over MIDI

A MIDI Out channel, added with **Add channel > MIDI Out (external synth)**, is the classic way: a channel of its own that sends its notes to the synth. It is also a trap. Plug in a synth, pick it under **MIDI > Output**, open a demo, press Play... and if nothing were done about it, the synth would say nothing, because no demo has a MIDI Out channel. So there are two more ways: every channel at once, and one channel from its instrument menu.

Every channel at once: routes

MIDI > Send every channel is on from the start. While an output is connected, every channel plays on the synth instead of the speakers, on the same MIDI channels that **File > Export song as MIDI...** writes, so a song sounds the same played live as from its MIDI file. A *route* is where one channel's notes go out:

```

pub type Route = (u8, Option<Note>);

pub fn routes(channels: &[Channel]) -> Vec<Option<Route>> {
    channels
        .iter()
        .zip(midi_channels(channels))
        .map(|(channel, midi_channel)| match &channel.kind {
            _ if channel.sent_over_midi().is_some() => None,
            ChannelKind::Sampler(SampleSource::Builtin(drum)) => {
                Some((midi_channel, Some(general_midi_drum(*drum))))
            }
            _ => Some((midi_channel, None)),
        })
        .collect()
}

```

A route is a MIDI channel, and the drum note if there is one. `type` only gives a long type a short name; a `Route` is still a plain tuple. `zip` walks two lists side by side: each channel with its MIDI channel, from `midi_channels` (below). The first arm, `_ if ...`, is a wildcard with a guard: it matches any kind, but only for a channel already sent over MIDI, which sends its notes itself and needs no route.

While every channel is sent, each sampler and synth is simply *replaced* by a `MidiOut`, and put back when it stops:

```

pub fn to_midi(self, channel: u8, pitch: Option<Note>, program:
Option<u8>) -> Self {
    match self {
        Instrument::MidiOut(_) => self,
        _ => Instrument::MidiOut(
            MidiOut::new(channel)
                .playing_as(pitch)
                .with_program(program),
        ),
    }
}

```

`to_midi` takes `self` by value: the old instrument is *moved* into the function and dropped there, and the caller gets the new one back. A MIDI Out already sends its notes, so it comes back unchanged.

The routes depend on the menu (an output picked, the box ticked) and on the rack: adding a synth above another one moves the lower one to the next MIDI channel. Rather than chase every place that can change them, the app works them out again every frame and compares:

```

let wanted = if self.live.send_all && self.live.midi.output().is_some() {
    midi::routes(channels)
} else {
    vec![None; channels.len()]
};
if self.live.routes_sent.as_ref() == Some(&wanted) {
    return;
}

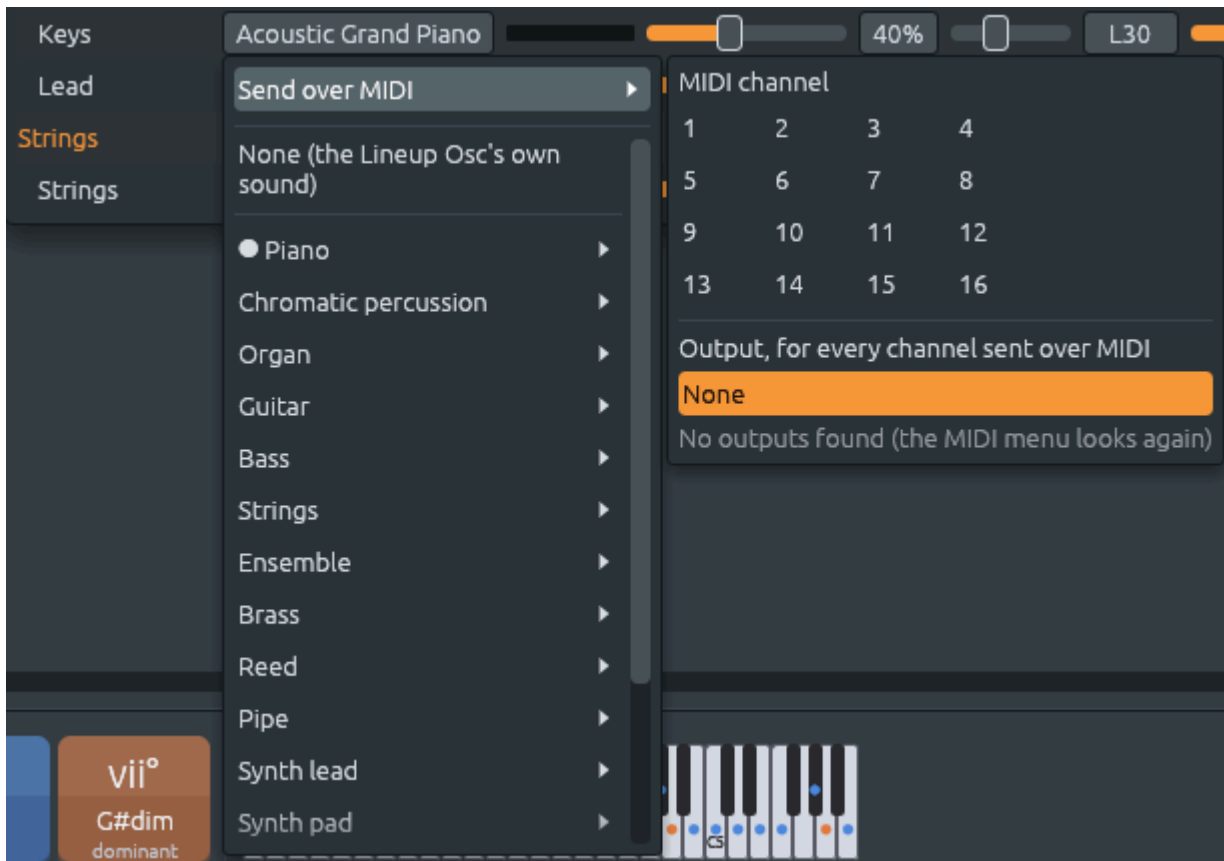
```

Working out 32 routes at most is far cheaper than drawing the window, and comparing two `Vec`s with `==` compares them element by element, because tuples, `Option` and `Note` all implement `PartialEq`. Only when they differ does a channel get a new instrument, a MIDI Out or its own, through `Command::SetInstrument`, and only the channels whose route changed.

`routes_sent` is an `Option<Vec<...>>`: `None` means “not known”. When channels are added, removed or reloaded, the audio thread gets plain instruments for them, and the ones that moved keep whatever they had. If any of them was a MIDI Out in place of its own instrument, the app forgets what it sent, and the next frame sends every channel again. That is rare (it takes a change to the rack), and it cannot go wrong.

One channel from its instrument menu

Often you want something in between: the bass on a hardware synth and the rest of the band in the app, or the drums on a drum machine. So every instrument menu, in the mixer and in the channel rack, starts with **Send over MIDI**: a MIDI channel from 1 to 16, and the output.



A synth sent over MIDI **becomes** a MIDI Out channel, keeping its instrument:

```
pub fn send_over_midi(&mut self, midi_channel: u8) {
    let midi_channel = midi_channel.min(15);
    match self.kind {
        ChannelKind::Sampler(SampleSource::Builtin(_)) => {
            self.sent_on = Some(midi_channel);
            return;
        }
        ChannelKind::Sampler(SampleSource::File(_)) => return,
        _ => {}
    }
    let plain = match self.kind {
        ChannelKind::Synth => self.name == Channel::synth().name,
        ChannelKind::MidiOut { channel } => self.name ==
Channel::midi_out(channel).name,
        _ => false,
    };
    if plain {
        self.name = Channel::midi_out(midi_channel).name;
    }
    self.kind = ChannelKind::MidiOut {
        channel: midi_channel,
    };
}
```

A name that only said what played it, “Lineup Osc” or “MIDI Out 2”, follows the change and becomes “MIDI Out 3”. A name you gave it, such as “Bass”, stays. `play_here` does the opposite. A sample from a file has no note a synth could play in its place, so it cannot be sent.

A **drum** does not change kind. It gets a field, `sent_on`, with the MIDI channel, and stays a `Sampler` playing its built-in drum. That matters to the band: styles find the kick, the snare and the hats by what they are (`Sampler(Builtin(Drum::Kick))`), to write their beats on them. A kick turned into a MIDI Out channel would no longer be found, and the next style would add a second kick. For the same reason the styles find a “Bass” or “Keys” sent over MIDI by its name, as the bass and the keys ([The library](#)). On the audio thread `Instrument::for_channel` builds a `MidiOut` for a drum with `sent_on`, playing its own note of a General MIDI drum set ([Instruments](#)), so the drum part, the band and the styles stay exactly as they were; only the sound goes elsewhere.

`Channel::sent_over_midi` answers “is it sent, and on which channel?” for both kinds at once, so the rest of the code does not have to know the difference.

The mixer names the drum a channel sends, from the General MIDI drum table in `general_midi.rs`:

```
/// The drum General MIDI plays for `note` on channel 10, such as "Bass
/// Drum 1" for 36, if the drum set has one there.
pub fn drum_name(note: u8) -> Option<&'static str> {
    DRUMS
        .get(usize::from(note.checked_sub(FIRST_DRUM)?))
        .copied()
}
```

Two kinds of “maybe” meet here. `checked_sub` returns `None` instead of wrapping around when the note is below 35 (a `u8` can’t go below 0), and the `?` returns that `None` from the function at once. `get` returns `None` past the end of the table, for notes above 81. `copied` turns the `Option<&&str>` that `get` gives into an `Option<&str>`.

The menu needs the list of outputs

The instrument menus are drawn by functions in `channel_rack.rs`, which know a channel and nothing about the app’s MIDI connections. Passing those down through the rack, the mixer and the style editor would add a parameter to many functions. Instead the app puts the list of outputs into egui’s memory once a frame, and the menu reads it there:

```
pub fn set_midi_outputs(ctx: &egui::Context, outputs: MidiOutputs) {
    ctx.data_mut(|data| data.insert_temp(midi_outputs_id(), outputs));
}
```

```
let outputs: MidiOutputs = ui
    .ctx()
    .data(|data| data.get_temp(midi_outputs_id()))
    .unwrap_or_default();
```

egui's memory can hold a value of any type that is `Clone`, `Send`, `Sync` and `'static`, one per `Id` and type. Values stored with `insert_temp` are not saved when the app closes, which is right for a list of ports that may be different tomorrow. The menu does not change the connection itself: it pushes a `RackAction::ConnectMidiOutput`, which the app carries out, as every other click in the rack.

A test of this menu, `the_mixer_sends_a_channel_over_midi_and_back`, shows why a test must ask the machine what it has. It opens **Send over MIDI** and checks the outputs listed. Expecting “No outputs found” passes on Linux and macOS test machines, which have no MIDI output, and fails on Windows, which always has one: its built-in “Microsoft GS Wavetable Synth”. So the test reads the outputs the app found, and expects that text only when there are none. It is the kind of difference running the tests on three systems is there to catch ([Three systems and continuous integration](#)).

Which MIDI channel each channel uses

A program reading a MIDI file, or a synth hearing the app, uses the MIDI channel to pick the instrument. General MIDI, the standard most synths follow, keeps **channel 10** (9, counted from 0) for drums, with a fixed note for each drum: 36 is a kick, 38 a snare, 39 a clap, 42 a closed hi-hat. One function decides the MIDI channel of every channel, for the MIDI file and for the routes alike:


```

pub fn midi_channels(channels: &[Channel]) -> Vec<u8> {
    let taken = |number: u8| {
        channels
            .iter()
            .any(|channel| channel.sent_over_midi() == Some(number))
    };
    let free: Vec<u8> = (0..16u8)
        .filter(|&number| number != DRUM_CHANNEL && !taken(number))
        .collect();
    let mut melodic = if free.is_empty() {
        (0..16u8).filter(|&number| number != DRUM_CHANNEL).collect()
    } else {
        free
    }
    .into_iter()
    .cycle();
    channels
        .iter()
        .map(|channel| match channel.kind {
            ChannelKind::Sampler(SampleSource::Builtin(_)) => {
                channel.sent_on.unwrap_or(DRUM_CHANNEL)
            }
            ChannelKind::MidiOut { channel } => channel,
            _ => melodic.next().unwrap_or(0),
        })
        .collect()
}

```

The built-in drums share channel 10, unless one was sent elsewhere. Channels sent over MIDI keep the channel they were given. Every other channel takes the next *free* one: not 10, and not one a channel sent over MIDI already plays on, or the two would share one sound on the synth and cut each other's notes short. `cycle` repeats the free channels for ever, and the iterator is created once, outside the closure, which borrows it mutably: each `next()` moves it on, so the channels get 1, 2, 3... in turn, and with more channels than free ones, they start sharing.

Writing a MIDI file

A Standard MIDI File holds the same messages, with times. It holds notes, not sound: "C4 starts here, ends there, on channel 1". The program that opens it chooses the instruments. That makes it the common language of music programs, and the way to take a Ferrous Lineup song into another studio. `song_to_midi` writes the song as the playlist plays it, and `pattern_to_midi` one pattern, once.

The file is a list of *chunks*. Each one starts with four letters and its length:

```
MThd  length 6  format 1  3 tracks  96 ticks per beat      the header
MTrk  length ... name "Ferrous Lineup", tempo, time signature, key, end
MTrk  length ... name "Lineup Osc", a program change, the notes, end
MTrk  length ... name "Kick", the notes of the kick, end
```

This is **format 1**: several tracks played together. The first track holds only the tempo, the time signature and the key, and every channel of the project gets a track of its own, named after it, that starts with a program change when the channel has a General MIDI instrument. A channel with no notes, or one that is not heard (muted, or silenced by another's solo), gets no track.

Numbers in a MIDI file are big-endian: the most significant byte first. Rust's integers have a method for that, which gives an array of bytes:

```
let mut file = Vec::new();
file.extend_from_slice(b"MThd");
file.extend_from_slice(&6u32.to_be_bytes()); // the header's length
file.extend_from_slice(&1u16.to_be_bytes()); // format 1
file.extend_from_slice(&(tracks.len() as u16).to_be_bytes());
file.extend_from_slice(&TICKS_PER_BEAT.to_be_bytes());
```

`b"MThd"` is a **byte string**: a `&[u8; 4]` with the ASCII codes of the letters, instead of a `&str`. `6u32` is the number 6 as a `u32` (the type written after the number), so `to_be_bytes` gives four bytes: `[0, 0, 0, 6]`. A `u16` gives two.

Ticks and delta times

Time in a MIDI file is counted in *ticks*, and the header says how many ticks make a beat. Ferrous Lineup writes 96, the same ticks its notes are placed in: a step, a sixteenth note, is 24 ticks, and a note's `nudge` and `trim` count the ticks it starts after or ends before its steps ([The project](#)). So a note between two steps is written exactly where it plays.

```
pub const TICKS_PER_BEAT: u16 = 96;
pub const TICKS_PER_STEP: u32 = crate::pattern::TICKS_PER_STEP as u32;

// A check made while compiling: if someone changed one of the two
// numbers and not the other, the build would stop here.
const _: () = assert!(TICKS_PER_BEAT as u32 == TICKS_PER_STEP *
    STEPS_PER_BEAT);
```

`const _: () = assert!(...)` is a constant with no name (`_`) whose value is worked out while compiling. `assert!` works in a `const` since Rust 1.57, and a failed assertion

there is a compile error, not a panic at run time. It costs nothing in the program, and it turns “remember to change both” into something the compiler checks.

Inside a track, each event is written after a *delta time*: the number of ticks since the event before it, not since the start. Most deltas are small (0 for the notes of a chord, 24 for the next step), so they are written in a format where small numbers take one byte, the **variable-length quantity**: seven bits of the number per byte, the most significant first, with the top bit set on every byte except the last.

```
200  = 1 1001000      -> 1000 0001 0100 1000  (0x81 0x48)
      |  L last 7 bits      |
last byte  L the rest: 1      L top bit clear: the
384  = 11 0000000      -> 0x83 0x00
                        L top bit set: more to come
```

```
fn write_variable_length(out: &mut Vec<u8>, value: u32) {
    let mut groups = [0u8; 5];
    let mut count = 0;
    let mut rest = value;
    loop {
        groups[count] = (rest & 0x7F) as u8;
        count += 1;
        rest >>= 7;
        if rest == 0 {
            break;
        }
    }
    for index in (0..count).rev() {
        let more = if index > 0 { 0x80 } else { 0 };
        out.push(groups[index] | more);
    }
}
```

The loop cuts the number into groups of seven bits from the low end, because that is the end `& 0x7F` and `>>= 7` can reach. The file wants them from the high end, so the second loop walks the groups backwards with `(0..count).rev()`. Five groups are enough for any `u32`, so they fit in an array, and nothing is allocated. The test `variable_length_numbers` feeds it the examples printed in the MIDI file specification.

A `TrackWriter` keeps the tick of the last event, so the rest of the code can think in absolute ticks:

```

#[derive(Default)]
struct TrackWriter {
    data: Vec<u8>,
    last_tick: u32,
}

impl TrackWriter {
    /// Writes the time since the last event. Events must come in order.
    fn delta(&mut self, tick: u32) {
        let delta = tick.saturating_sub(self.last_tick);
        write_variable_length(&mut self.data, delta);
        self.last_tick = self.last_tick.max(tick);
    }

    // ...

    /// Ends the track at `end` (or at its last event, if that is later)
    /// and wraps it in its chunk header.
    fn finish(mut self, end: u32) -> Vec<u8> {
        self.meta(end.max(self.last_tick), END_OF_TRACK, &[]);
        let mut chunk = Vec::with_capacity(8 + self.data.len());
        chunk.extend_from_slice(b"MTrk");
        chunk.extend_from_slice(&(self.data.len() as u32).to_be_bytes());
        chunk.extend(self.data);
        chunk
    }
}

```

`#[derive(Default)]` gives `TrackWriter::default()`, an empty `Vec` and tick 0. `finish` takes `mut self`, not `&mut self`: it consumes the writer, and its data moves into the chunk without a copy. A finished writer cannot be written to by mistake, because it no longer exists.

Meta events: names, tempo, time and key

Events that are not notes start with `0xFF`, a kind, and a length:

Bytes	Means
FF 03 len text	the track's name
FF 51 03 tt tt tt	the tempo, as microseconds per beat
FF 58 04 nn dd cc 08	the time signature: 4/4 is 04 02 18 08
FF 59 02 sf mi	the key signature: sharps (negative for flats), major or minor
FF 7F len data	data for one program only (sequencer-specific)

Bytes	Means
FF 2F 00	the end of the track

The tempo is the odd one: not beats per minute, but how long one beat lasts, in microseconds. A song whose tempo changes gets a tempo event on every step where the played tempo changes, and the song's time signatures go in at every change; both are in [Tempo, meter, swing and feel](#).

The key signature says how many sharps or flats, and whether the key is minor.

`Scale::key_signature` works it out by going round the circle of fifths ([Scales, modes and keys](#)). A key signature only knows major and minor, so a mode takes the signature of the major key it comes from: D Dorian has none, like C major, and E Dorian the two sharps of D major. That loses something: a file that says "two sharps, major" cannot say whether it means D major or E Dorian. So next to each key signature the file gets the exact key, in a **sequencer-specific** event, the one place the standard leaves for a program's own data:

```
/// The start of our own event: 0x7D is the maker's number set aside for
/// programs that are not sold, then our name, so another program's
/// 0x7D event is never mistaken for ours.
const SCALE_EVENT_START: &[u8] = b"\x7DFerrous Lineup key ";

pub fn scale_event(scale: Scale) -> Vec<u8> {
    let mut data = SCALE_EVENT_START.to_vec();

    data.extend_from_slice(ron::to_string(&scale).unwrap_or_default().as_bytes());
    data
}

/// The key a sequencer-specific event holds, if [scale_event] wrote it.
pub fn scale_from_event(data: &[u8]) -> Option<Scale> {
    let key = data.strip_prefix(SCALE_EVENT_START)?;
    let text = std::str::from_utf8(key).ok()?;
    ron::from_str(text).ok()
}
```

The key is written the way a project file writes it, in RON, so the event reads `7D "Ferrous Lineup key (root:E,kind:Dorian)"`, and the code that saves projects reads it back ([Saving](#)). `\x7D` in a byte string is the byte 0x7D. `strip_prefix` gives the rest of the slice after the start, or `None` when the slice does not start that way; the `?`s make every step that can fail end the function with `None`. Other programs skip the event, and Ferrous Lineup reads back E Dorian, C# harmonic minor or a twelve-tone row exactly. A row or a set has no key signature, so for those this event is all there is.

Notes, drums and order

Each note becomes two events, its note-on and its note-off. The notes are the ones playback plays: `played_notes` gives them with the song's swing, `song_steps_of` every song step where a piece of the song plays that step (once per repeat), and a note running past the end of its pattern, or of the piece being played, is cut there, exactly as the audio thread plays it:

```
let room = (length - note.start).min(play.end() - song_step);
let step = song_step as u32 * TICKS_PER_STEP;
let on = step + u32::from(note.nudge);
let off = if note.length <= room {
    on + note.ticks() as u32
} else {
    step + room as u32 * TICKS_PER_STEP
};
let pitch = drum.map_or(note.pitch, general_midi_drum);
```

`map_or(note.pitch, general_midi_drum)` gives the note's own pitch for a melodic channel, and for a built-in drum, its General MIDI note: a drum's steps are all written on C4, which a General MIDI drum set plays as a bongo. Passing the function `general_midi_drum` itself, by name, works because its type, `fn(Drum) -> Note`, is what `map_or` asks for.

Then the events are sorted by time. At the same tick, note-offs must come before note-ons: if a note ends exactly where the same note starts again, "on, off" would end the new note at once, and only "off, on" is heard twice.

```
events.sort_by_key(|&(tick, message)| (tick, matches!(message,
MidiMessage::NoteOn { .. })));
```

The key is a tuple, `(tick, is_note_on)`, and tuples compare item by item. For the same tick, `false < true`, so note-offs win. `sort_by_key` is a *stable* sort: events with equal keys keep their order, so the notes of a chord stay in the order they were written. The test `a_repeated_note_ends_before_it_starts_again` checks the order at the tick where a C4 ends and starts again.

In the app, **File > Export song as MIDI...** and **Export pattern as MIDI...** only ask where to save, then write the bytes with `std::fs::write`. A pattern's name could hold a `/`, which a file name cannot, so the name offered for a pattern replaces it.

Reading a MIDI file

File > Import MIDI file..., opening a `.mid` file from the command line, or dropping one on the window turns a MIDI file into a new project, ready to play. MIDI files are the easiest way to get real music into the app: almost every piece ever written can be found as one, often typed in from the score, sometimes played in on a keyboard.

Reading is writing walked backwards, with one difference that changes how the code is written: **a file we read can be wrong**. It may be cut short by a broken download, it may claim a chunk is longer than it is, it may not be a MIDI file at all. Whatever the bytes are, the program must not crash.

A cursor that cannot fall off the end

The reader walks the bytes with a small *cursor*: a slice and a position in it.

```
struct Bytes<'a> {
    data: &'a [u8],
    at: usize,
}

impl<'a> Bytes<'a> {
    // ...

    fn byte(&mut self) -> Option<u8> {
        let byte = *self.data.get(self.at)?;
        self.at += 1;
        Some(byte)
    }

    /// The next `count` bytes, or `None` if there are fewer left.
    fn take(&mut self, count: usize) -> Option<&'a [u8]> {
        let end = self.at.checked_add(count)?;
        let slice = self.data.get(self.at..end)?;
        self.at = end;
        Some(slice)
    }
}
```

Two Rust ideas make this safe:

- **'a is a lifetime.** `Bytes<'a>` borrows the file's bytes; it does not copy them. The lifetime says "a `Bytes` cannot outlive the bytes it points into", and the compiler checks that. `take` returns `&'a [u8]`: a slice of the *original* bytes, which stays valid even after the cursor moves on.

- **get instead of indexing.** `data[at]` panics when `at` is past the end; `data.get(at)` returns `None`. The `?` after it returns that `None` from `byte` straight away. `checked_add` does the same for a length so large that adding it would overflow. So every read either gives a value or `None`, and the caller decides what a missing value means.

A delta time is read back by gluing the groups of seven bits together:

```
fn variable_length(&mut self) -> Option<u32> {
    let mut value = 0u32;
    for _ in 0..4 {
        let byte = self.byte()?;
        value = (value << 7) | u32::from(byte & 0x7F);
        if byte & 0x80 == 0 {
            return Some(value);
        }
    }
    None
}
```

`81 00` is `1 << 7 | 0 = 128`. The standard allows four bytes at most; a fifth would mean the file is damaged, so after four the loop gives up.

The header and the chunks

`read` checks for `MThd`, reads the format and the *division*, and then reads every chunk to the end of the file:


```

while let Some(id) = file.take(4) {
    let Some(length) = file.u32() else {
        damaged = true;
        break;
    };
    let length = length as usize;
    // A chunk that claims to be longer than what is left of the file
    // was cut short: read what there is.
    let data = file.take(length.min(file.remaining()));
    let data = data.unwrap_or_default();
    if data.len() < length {
        damaged = true;
    }
    // Chunks of other kinds may be added by future versions of the
    // standard, or by one program for itself: they are skipped.
    if id == b"MTrk" {
        let (track, complete) = read_track(data);
        damaged |= !complete;
        tracks.push(track);
    }
}

```

The header also says how many tracks there are, but some programs write that number wrong, so the chunks themselves are trusted instead. Real files are often slightly broken, and every program that opens them is forgiving about it. So is this one: a track cut short keeps the events read so far, a chunk longer than the file is read as far as it goes, unknown chunks are skipped. But the result says so:

`MidiFile::damaged` is set, and the notice after the import ends with “The file is damaged: everything before the damage was read”. `damaged |= !complete` is the boolean “or” written as an assignment, like `+=` : once damaged, always damaged.

Two more things a reader meets in the wild:

- **RMID files** (`.rmi`). Windows used to wrap MIDI files in RIFF, the container of WAV files: `RIFF` , a size, `RMID` , then chunks. The MIDI file is the chunk called `data` . Unlike MIDI, RIFF writes its numbers least significant byte first, so `unwrap_riff` uses `u32::from_le_bytes` where MIDI uses `u32::from_be_bytes` , and a chunk of odd length is followed by a byte of padding.
- **Ticks per second**. The division usually counts ticks per beat (480 and 960 are common). With its top bit set it counts ticks per frame of film or video instead (24, 25, 29.97 or 30 frames a second), as `Timing::PerSecond` . Such a file has no beats at all; the import counts its time in beats of 120 BPM, so it plays at the right speed.

Running status

A track of a piano piece is mostly note on, note off, note on... To save space, a file may leave out the status byte when it is the same as the one before. The reader can always tell, because a status byte has its top bit set and a data byte does not:

```
let (status, first_data) = if first & 0x80 != 0 {
    (first, bytes.byte())
} else {
    (running.unwrap_or(0), Some(first))
};
```

So `90 3C 64 00 40 64` is two notes on: C4 and E4, both at velocity 100. Program changes and channel pressure have one data byte, the rest of the channel messages two. A meta event (`FF`) and a system exclusive message (`F0` or `F7`, settings for one maker's synths, which are skipped) carry their own length. Any other system message only travels down cables, never in files, so meeting one means the track is damaged.

Events as an enum

Each event the import needs becomes a variant of one enum, with its data inside:

```
pub enum EventKind {
    NoteOn { channel: u8, key: u8, velocity: u8 },
    NoteOff { channel: u8, key: u8 },
    Controller { channel: u8, number: u8, value: u8 },
    Program { channel: u8, program: u8 },
    Tempo(u32),
    TimeSignature { beats: u8, unit: u8 },
    KeySignature { sharps: i8, minor: bool },
    ExactKey(Scale),
    TrackName(String),
    InstrumentName(String),
}
```

Everything else (pitch bend, aftertouch, lyrics) is read past and dropped. A note-on with velocity 0 becomes a `NoteOff`, as on a cable. `meta_event` uses slice patterns to check that a meta event's data has the right shape before trusting it:

```
// Sharps (negative for flats), then 0 for major or 1 for minor.
(KEY_SIGNATURE, &[sharps, mode]) => {
    let sharps = sharps as i8;
    ((-7..=7).contains(&sharps) && mode <=
1).then_some(EventKind::KeySignature {
    sharps,
    minor: mode == 1,
})
}
```

A key signature is exactly two bytes. `&[sharps, mode]` only matches a slice of length two and names its bytes; a key signature of three bytes, or one claiming 9 sharps, is not a key signature and is skipped. `sharps as i8` reads the same eight bits as a signed number, so the byte `0xFE` is -2: two flats. `bool::then_some` gives `Some(value)` when the `bool` is true and `None` otherwise.

The names of tracks and instruments are text, and the standard says nothing about how letters beyond plain English are written: newer programs use UTF-8, older ones Latin-1, one byte per letter. `text` tries UTF-8 first, since Latin-1 text is almost never valid UTF-8, and otherwise turns each byte into the `char` with that number, which is exactly Latin-1.

Importing: from events to a project

`import(&file, title)` turns a `MidiFile` into a `Project` and a `Report` of what it did. The report becomes the notice in the status bar, such as *"Clementi: 2 channels, 812 notes, 38 bars, 4/4, 120 BPM, C major"*.

In the app, reading the file, reading the MIDI and importing are three steps that can each fail with a different error type. They are chained into one `Result`:

```
let imported = std::fs::read(path)
    .map_err(|err| err.to_string())
    .and_then(|bytes| midi_file::read::read(&bytes).map_err(|err|
err.to_string()))
    .and_then(|file| {
        midi_file::import::import(&file, &title).map_err(|err|
err.to_string())
    });
```

`and_then` runs the next step only if the one before succeeded, and passes its value on. `and_then` keeps the error type, so each step turns its own error into a `String` with `map_err` first. One `match` at the end shows the notice or the error. Reading

takes a few thousandths of a second, so unlike `importing a recording`, it is done straight away on the UI thread.

Ticks into our ticks

A file counts in its own ticks, often 480 or 960 to the beat; a project counts 96. `Clock` turns one into the other, to the nearest tick:

```
Timing::PerBeat(per_beat) => {  
    let per_beat = u128::from(per_beat);  
    (u128::from(tick) * u128::from(TICKS_PER_BEAT) + per_beat / 2) /  
    per_beat  
}
```

Adding half the divisor before dividing rounds to the nearest integer instead of down. The sum is done in `u128` because a damaged file can claim a tick near `u64::MAX`, and multiplying that by 96 would overflow a `u64`; the result is then capped, far past the end of any song.

Notes between steps

A pattern counts time in steps. Rounding every note to the nearest step would ruin a lot of music:

- **Triplets** split a beat in three. Eighth-note triplets start at 0, $\frac{1}{3}$ and $\frac{2}{3}$ of a beat: none of those is a sixteenth.
- **Swing** plays the second eighth of each beat late.
- A file **played in** on a keyboard has every note a little early or late, and that is a large part of what makes it sound alive.

So a note keeps its exact ticks in `nudge` and `trim`. Why 24 to the step? Because 24 divides by 2, 3, 4, 6, 8 and 12. An eighth-note triplet is 32 ticks long, a sixteenth triplet 16: both whole numbers, so triplets land *exactly*. `NoteEvent::exact` builds a note from its exact ticks:

```
pub fn exact(pitch: Note, start: usize, ticks: usize) -> Self {
    let end = start + ticks.max(1);
    let first = start / TICKS_PER_STEP;
    let last = end.div_ceil(TICKS_PER_STEP);
    NoteEvent {
        nudge: (start % TICKS_PER_STEP) as u8,
        trim: (last * TICKS_PER_STEP - end) as u8,
        ..NoteEvent::new(pitch, first, last - first)
    }
}
```

The triplet on tick 32 is in step 1 ($32 / 24$), nudged by 8 ticks ($32 \% 24$). `div_ceil` divides rounding up, so `last` is the step after the one the note ends in. The piano roll draws a nudged note where it really starts, the audio thread plays it there ([The sequencer](#)), and **Tools > Quantize to the steps** moves notes onto the grid when you want them there.



Pairing note on with note off

A file does not store notes. It stores *key down* and *key up*, and the import has to pair them. While a track is read, each MIDI channel keeps the notes still sounding on each key:

```
struct ChannelState {
    /// For each key, the notes sounding on it, oldest first. There is
    /// usually one at most, but a file can strike a key again before
    /// letting it go; each note off then ends the oldest note.
    open: Vec<Vec<Open>>,
    pedal: bool,
    notes: Vec<Played>,
}
```

A key can have two notes open: two voices of a fugue on the same note, or a round where the parts overlap. Then each key up ends the **oldest** note on that key, first in, first out. Ending the newest one instead would cut notes short wherever parts overlap: the rounds among the demos would not come back as they were written.

Two more things the import does like a piano would:

- **The sustain pedal** (controller 64). While it is down, a key let go keeps sounding (`held_by_pedal`); the note ends when the pedal comes up, or when the same key is struck again. Without this, a piano piece played in with the pedal sounds dry and choppy, because the player's fingers left the keys long before the sound was meant to stop.
- **A note let go on the tick it started** (a grace note, a drum hit) still lasts one tick, so it is heard.

Lifting the pedal splits each key's list in two:

```
let (held, sounding): (Vec<Open>, Vec<Open>) =
    open.drain(..).partition(|note| note.held_by_pedal);
*open = sounding;
for note in held {
    self.end(key, note, tick);
}
```

`partition` sends each item to one of two collections, depending on the closure, and keeps their order. `drain(..)` empties the list as it goes, so the notes are moved, not copied, and `*open = sounding` puts the ones still sounding back. Keys still down when a track ends are let go there.

A note struck twice on the same tick by mistake is one note: the notes are sorted with `sort_by_key(|note| (note.start, note.pitch, std::cmp::Reverse(note.end)))`, and `dedup_by_key` keeps the first of each `(start, pitch)`. `Reverse` turns an order round, so the longer of the two comes first and is the one kept.

Tracks, channels and drums

Each MIDI channel of each track that plays notes becomes a channel of the rack. Most files are **format 1**, with one track per instrument, so each track becomes a channel, named after the track (“Violin I”), or after its General MIDI instrument when the track has no name. A **format 0** file puts everything in one track, with the instruments on different MIDI channels; there, each MIDI channel becomes a rack channel. A rack holds 32 channels; when a file has more parts than fit next to its drums, its parts are joined by MIDI channel.

The **program change** that chose each part’s instrument is kept as the channel’s `program` ([SoundFonts and General MIDI](#)), and its **volume** (controller 7) sets the channel’s volume, so a quiet accompaniment stays quiet. Programs and volumes are sometimes set on another track than the notes, so the first one set on each channel anywhere in the file is the fallback:

```
EventKind::Program { channel, program } => {  
    programs[usize::from(channel)].get_or_insert(program);  
}
```

`Option::get_or_insert` sets the value only if there is none yet, so the first program change wins without an `if`.

MIDI channel 10 is the drum channel: each key is a drum sound. The import gives each built-in drum the file uses a channel of its own, as in a new project, so the drums show up in the channel rack as a step sequencer, each hit one step long. Sounds the built-in drums do not have go to the closest one (`general_midi_drum_sound`: a triangle to the ride, a tambourine to the shaker, bongos to the conga); a sound with nothing close, such as a whistle, is left out, and the notice counts it.

A few files on the internet have millions of notes, written as a stunt, and would make the piano roll too slow to use. The import keeps the 50,000 earliest (`MAX_NOTES`). `keep_first_notes` sorts the start ticks, takes the start of the last note that fits as the cut, and uses `partition_point` (a binary search for the first item that fails a test) to count how many notes on the cut tick still fit.

Tempo, time and key

Tempo changes, time signatures and key signatures apply to the whole song, whichever track holds them. One generic function gathers events of one kind from every track:

```

fn song_events<T>(tracks: &[Track], pick: impl Fn(&EventKind) ->
Option<T>) -> Vec<(u64, T)> {
    let mut events: Vec<(u64, T)> = tracks
        .iter()
        .flat_map(|track| &track.events)
        .filter_map(|event| pick(&event.kind).map(|value| (event.tick,
value)))
        .collect();
    // A stable sort keeps events on the same tick in the order they were
    // read, so the last one written wins.
    events.sort_by_key(|&(tick, _)| tick);
    events
}

```

`pick` says which events to keep and what to take from them, so the same function serves tempos (`u32`), time signatures and keys. Each kind then goes into a `BTreeMap` by step, where a later event on the same step replaces an earlier one, as on playback. Every tempo change becomes a tempo marker, every time signature a time marker (`Tempo, meter, swing and feel`), and `tidy` leaves out the markers that change nothing: files often repeat the time signature at the start of every section. A time signature the app cannot count, with bars that are not a whole number of sixteenths, is reduced when it can be: 6/32 becomes 3/16.

The key comes from the key signatures, as written: two flats and minor is G minor, and the signature's sharps or flats decide the spelling, so six flats are G $\flat$ major and six sharps F $\sharp$ major. Every later signature that names a different key becomes a key marker on the playlist. The exact key from a file Ferrous Lineup wrote wins over the signature next to it. Two cases are left to the notes:

- **A file with no key signature at all.**
- **A file that says C major but is not.** Many programs write “no sharps, major” into every file, whether or not anyone chose a key. So when every key signature in the file says C major, the import checks it against the notes, and if another key fits *clearly* better (by `CLEARLY_BETTER`, 0.1 of correlation) it takes that one, and the notice says “worked out from the notes”.

How notes decide a key, with key profiles and the Krumhansl-Schmuckler method, is in [Hearing harmony](#). The import adds up how long each of the twelve pitch classes sounds, and asks `key_fits` which key fits best.

Cutting the song into patterns

A pattern holds at most 256 steps (16 bars of 4/4), and a song may last 200 bars. So the import cuts the song into patterns at **bar lines**, at most 16 bars each, and lays

them one after another on the playlist.

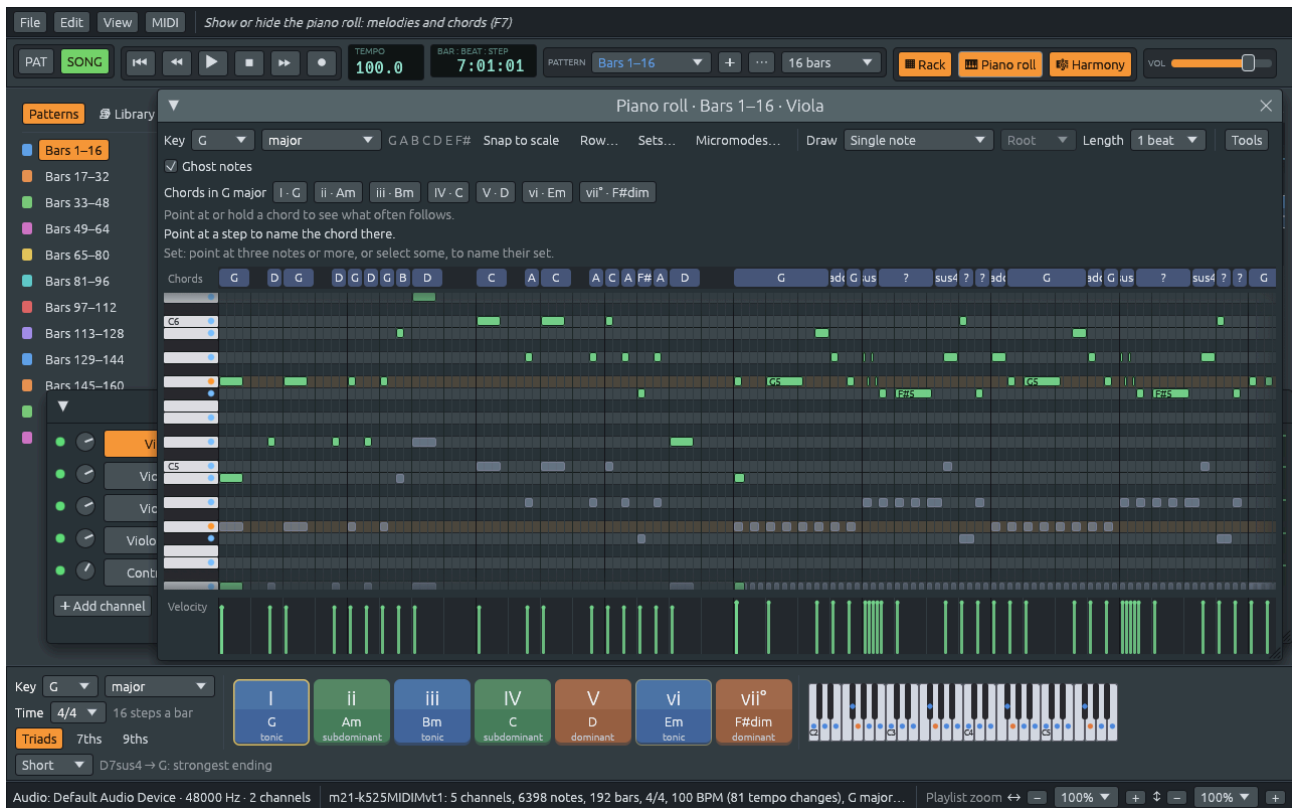
Where to cut? A note that sounds across the cut has to be split in two, which is harmless for the sound but ugly to edit. So among the bar lines in the second half of each stretch, `chunks` picks the one the fewest notes sound across, the latest of those. To count them for every step at once, it uses a **difference array**:

```
// How many notes cross each step's start: a difference array, +1
// where a note starts crossing and -1 after it stops.
let mut change = vec![0i64; length + 2];
for note in notes {
    let first = note.start / TICKS_PER_STEP + 1;
    let last = note.end.div_ceil(TICKS_PER_STEP).saturating_sub(1);
    if first <= last && first <= length {
        change[first] += 1;
        change[(last + 1).min(length + 1)] -= 1;
    }
}
let mut crossing = Vec::with_capacity(length + 1);
let mut count = 0;
for &delta in change.iter().take(length + 1) {
    count += delta;
    crossing.push(count);
}
```

For each note, +1 on the first step boundary it crosses and -1 after the last; adding the numbers up from the left gives, at every step, how many notes are sounding across its start. That is one pass over the notes and one over the steps, instead of checking every note at every step. The choice is then

`(earliest..=last).rev().min_by_key(...): min_by_key` returns the *first* of several equal minimums, and walking the bars backwards makes the first one the latest bar.

`place_patterns` fills the patterns, splitting the notes that cross a cut. A stretch that comes back **note for note** (a repeated verse, a da capo) plays the same pattern again instead of a copy, as you would build the song by hand. Each pattern is named after its bars, such as “Bars 1–16”.



Because the notes land in ordinary patterns, everything that already understands notes works on them: the piano roll's **Chords** lane names the chord at every step, and the harmony keys light up the chord sounding now while the song plays.

Testing it

- `midi.rs`: `parse` on note-ons, note-offs, a silent note-on, and messages it must ignore (a sustain pedal, a pitch bend, a clock tick, a message too short, a data byte out of range), and every message to bytes and back.
- `record.rs`: the quantizing rules, with made-up positions.
- `engine.rs`: `keys_play_the_live_channel_until_they_come_up`, `a_key_comes_up_on_the_channel_it_went_down_on`, `a_key_up_leaves_the_sequencers_notes_alone`, `keys_from_a_midi_keyboard_skip_the_command_queue`, `removing_channels_lets_go_of_their_keys`, `midi_out_channels_send_their_notes`, `removing_a_midi_out_ends_its_notes`, and the two tests that a MIDI message waits for room instead of being dropped, in order.
- `instrument.rs`: note-offs after the gate, a repeated note, the program change before the first note and after every stop, a synth or drum sent to MIDI that is silent and sends its notes.

- `midi_file.rs`: the variable-length numbers from the specification, then whole files, read back by a small reader written in the test module: tracks, program changes, tempo changes, time and key signatures, the exact key, drums on channel 10, clips and repeats, a repeated note, a section played once, and the MIDI channels of every kind of channel.
- `read.rs`: running status, meta events, system exclusive, an RMID file, ticks per second, and `a_file_cut_short_keeps_what_was_read`, which cuts a file at *every* length from 0 bytes to the whole file and reads each one without a panic.
- `import.rs`: small files built by hand for what a demo never has: triplets, the sustain pedal, a format 0 file, drums, keys with and without signatures, a long song cut at bar lines, a stretch that comes back, a file without notes.

The best test of an import is an export. `every_demo_comes_back_note_for_note` exports each song demo as a MIDI file, imports it, exports the result again, and compares the two files event by event: every note on with its key and velocity, every note off, every tempo change, time signature, key signature and program change, each at the same tick. It also checks the imported project has exactly the demo's key and key markers.

A reader written by the same person who wrote the writer can share its mistakes, though. So it is worth opening exported files in a program that knows nothing about Ferrous Lineup (Python's `mido` library reads one in a few lines), and files from other programs in the app.

Testing without a synth

A test can check the bytes an instrument would send, but not that they reach a port. For that, Linux has the ALSA sequencer, and two kernel modules that make pretend MIDI devices: `snd-seq-dummy` (a "Midi Through" port that passes everything on) and `snd-virmidi` (ports that look like a USB keyboard or synth). `aseqdump` prints every message that reaches a port:

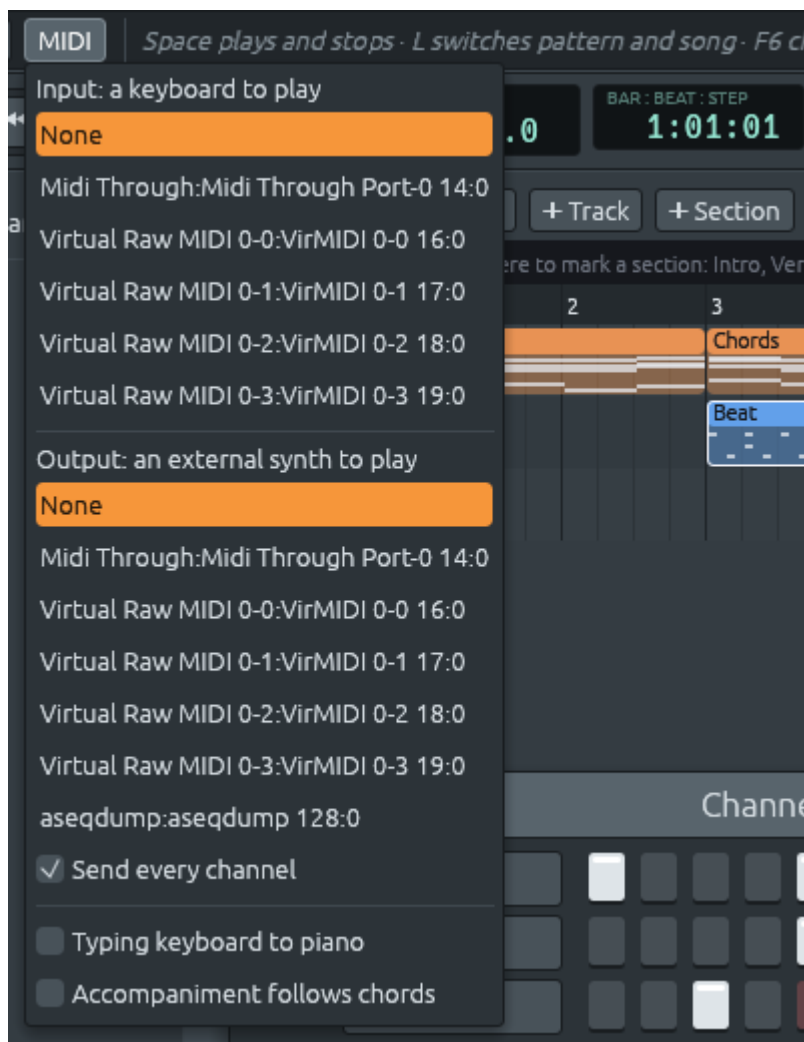
```
sudo modprobe snd-seq-dummy snd-virmidi
aseqdump -p "Midi Through"
```

Pick **Midi Through** under **MIDI > Output**, press Play, and each note appears as it is sent:

```
129:0    Note on          9, note 36, velocity 100
129:0    Note on          0, note 60, velocity 100
129:0    Note off         9, note 36, velocity 64
```

`aseqdump` counts channels from 0, so its 9 is MIDI channel 10, the drums. Writing bytes to a virtual port's raw device plays a key on a pretend keyboard: pick **Virtual Raw MIDI 0-0** as the input, then

```
printf '\x90\x45\x50' > /dev/snd/midiC0D0 # A4 down, velocity 80
printf '\x80\x45\x00' > /dev/snd/midiC0D0 # and up
```



Exporting: WAV, MIDI and video

Files: `crates/lineup-audio/src/render.rs`, `crates/lineup-app/src/app/export.rs`, `crates/lineup-video/`, `crates/lineup-app/src/video.rs` and `crates/lineup-app/src/app/video.rs`

The **File** menu takes a song out of the app in several forms:

- **Export WAV...** writes the sound, as you hear it when the song plays.
- **Export video...** writes an MP4 file with the sound and the views you pick following the music: the chord sheet, the lead sheet, the playlist, the piano roll, the harmony keys.
- **Export song as MIDI...** and **Export pattern as MIDI...** write the notes, for another music program. How the bytes of a MIDI file are made is in [MIDI](#).
- **Export chord sheet...** writes the chords as text, and the lead sheet prints as a PDF; both are in the chapters on the [chord sheet](#) and the [lead sheet](#).

How to use them is in the manual's [Exporting](#). This chapter is about the two exports that have to *play* the song: the WAV file and the video. Three ideas carry them:

1. **The same code, faster than real time.** The audio thread is a function that fills a buffer. It does not know a sound card is the one asking. A loop can ask instead, as fast as the computer can go, and write what comes out to a file. There is no second engine to keep in agreement with the first, so an export sounds exactly like playback.
2. **A snapshot, on a thread of its own.** The export gets its own copy of the song and its own instruments, and runs beside the window. You can keep playing and editing; the export goes on with the song as it was when you chose it.
3. **A finished file or none.** Every export writes to a temporary file and renames it at the end, so a failed or stopped export never leaves half a file where a finished one should be.

The audio thread, without a sound card

Seen from the outside, the audio thread is one method:

```
pub(crate) fn process<T>(&mut self, output: &mut [T])
where
    T: SizedSample + FromSample<f32>,
```

It fills `output` with the next frames, in whatever sample type the [sound card wants](#). With a sound card, `cpal` calls it a few hundred frames at a time, as fast as they are played. Nothing in it waits for the clock: it simply makes the next frames. So a loop that calls it with a plain `Vec<f32>` gets the song as fast as the processor can compute it. The engine's own tests have always worked this way: they never open a sound card.

`lineup-audio` wraps this up in a `Renderer`:

```
pub struct Renderer {
    audio: AudioThread,
    sample_rate: u32,
    channels: usize,
    /// The audio thread is built with a command queue, but the renderer
    /// hands it commands directly. Keeping this end of the queue alive
    /// keeps the queue valid.
    _commands: rtrb::Producer<Command>,
    trash: rtrb::Consumer<Garbage>,
    /// Where playback is, which the audio thread stores after every
    block.
    position: Arc<AtomicU64>,
    /// The trace being written, and its reader, once [Renderer::trace`]
    /// has started one.
    trace: Option<(TraceReader, Trace)>,
}
```

`AudioThread` is `pub(crate)`: other crates cannot build one, call `process` on it or reach into it. They get the `Renderer`, whose methods keep the audio thread's rules. Keeping the export *inside* `lineup-audio` is what lets the engine stay private.

`AudioThread::new` wants the receiving end of a command queue, because on the real audio thread that is how commands arrive. The renderer has no other thread to receive from. It makes a queue with room for one command:

```
let (commands, command_consumer) = rtrb::RingBuffer::new(1);
```

and never uses it. Commands go straight in:

```
/// Applies a command at once: there is no other thread to hand it to.
/// Volume changes take effect at once too, without smoothing: the
/// commands come before the first sample is made.
pub fn send(&mut self, command: Command) {
    self.audio.apply(command);
    self.audio.settle_volumes();
    self.empty_trash();
    self.read_trace();
}
```

The field holding the queue's other end is `_commands`. A leading underscore tells the compiler (and the reader) that the field is never read on purpose, so it does not warn "field `commands` is never read". The field still holds the producer for as long as the renderer lives, so the audio thread's queue has both its ends, as it does in the app.

The other methods are as short:

```
/// Fills `output` with the next frames, one sample per channel each.
pub fn render(&mut self, output: &mut [f32]) {
    self.audio.process(output);
    self.empty_trash();
    self.read_trace();
}
```

`read_trace` belongs to the [playback trace](#), which uses the same renderer to write down every note instead of the sound.

Two details that only show up offline

Smoothing. Volume changes glide over a few milliseconds (`Smoothed`), because a sudden jump makes an audible click. On a sound card, that glide happens while the music plays. In an export, setting the master volume just before the first sample would make the first milliseconds of the file glide from the default volume to yours. So `send` calls `settle_volumes`, which puts every smoothed value at its target at once:

```
pub(crate) fn settle_volumes(&mut self) {
    self.master_volume.settle();
    self.reverb.settle();
    for channel in &mut self.channels {
        channel.volume.settle();
        channel.pan.settle();
        channel.reverb_send.settle();
    }
}
```

```
/// Jumps straight to the target, with no smoothing.
pub fn settle(&mut self) {
    self.current = self.target;
}
```

The renderer's first test, `renders_like_the_audio_thread`, would catch a missing settle: it sets the master volume and checks that the very first frame of a click is at full level.

The trash. The real audio thread never frees memory. What it throws away (an old list of notes, a removed channel) goes into the [trash queue](#), and the UI thread frees it once per frame. The renderer is not a real-time thread, so it can free things straight away, and empties the queue after every command and every block:

```
/// Frees what the audio thread threw away. There is no real-time
/// thread here, so it can be done straight away.
fn empty_trash(&mut self) {
    while self.trash.pop().is_ok() {}
}
```

`pop` hands back each value, and dropping it frees it. The loop body is empty because there is nothing else to do with them.

Putting the song into the renderer

An export builds its renderer with the same commands the app sends at startup and after opening a file:


```

/// The whole song, ready for `renderer` to play from the start.
fn song_render_with(&mut self, mut renderer: Renderer) -> SongRender {
    let sample_rate = renderer.sample_rate();
    for command in self.project_commands(sample_rate, 1.0) {
        renderer.send(command);
    }

    // The song loops when it reaches its end, so silent steps are
    // added after it: enough for the tail, and one more, so the
    // export ends before the song would start again.
    // The whole song, every repeat included, even when song mode only
    // loops part of it.
    let song = self.project.arrangement();
    let song_length = song.length;
    // The silent steps play at the tempo the song ends at: the
    // audio thread plays steps past the end of the tempo list at
    // its last tempo.
    let tempos = self.project.played_tempos(&song);
    let end_tempo = tempos.last().copied().unwrap_or(self.project.tempo);
    let step_frames = frames_per_step(end_tempo, sample_rate);
    let reverb = self.project.mixer.reverb;
    let tail_seconds = if reverb.enabled {
        TAIL_SECONDS + f64::from(reverb.decay + reverb.pre_delay_ms /
1000.0)
    } else {
        TAIL_SECONDS
    };
    let tail_frames = tail_seconds * f64::from(sample_rate);
    let silent_steps = (tail_frames / step_frames).ceil() as usize + 1;
    let song_frames = self.project.played_seconds(&song) *
f64::from(sample_rate);
    renderer.send(Command::SetSong {
        plays: song.plays,
        length: song_length + silent_steps,
    });
    renderer.send(Command::SetSongTempos(tempos));
    renderer.send(Command::SetMode(PlayMode::Song));
    renderer.send(Command::Play);
    SongRender {
        renderer,
        frames: (song_frames + tail_frames) as usize,
    }
}

```

`project_commands` lists the tempo and loose timing, the master volume and reverb, every pattern (`InsertPattern`), every channel (`InsertChannel`, with its instrument and its notes in all patterns, then its pan, reverb send and effects) and the song. The second argument is the practice speed. Playback can slow the song down for practice, but an export is always at the song's own speed, so it passes `1.0`.

`send_whole_project`, which talks to the real audio thread, calls the same function with the practice speed, so both are built the same way.

What the export plays follows from those commands:

- **Everything you hear:** the band, the mixer's volumes, pans, reverb and effects. Muted channels and muted tracks stay silent, as they do when you press Play. With the [SoundFont](#) turned on, its instruments play.
- **The whole song,** every repeat and every time round a chorus, even when a loop region makes song mode play only part of it: `arrangement()` is the whole song, where Play uses `playback_arrangement()`.
- **No metronome.** `project_commands` never sends `Command::SetMetronome`, and a new audio thread starts with it off.

The instruments are made for the export's sample rate, 44,100 frames a second, whatever rate your sound card runs at. The built-in drums are synthesized again at that rate, the Lineup Osc is made for it, and samplers get their sounds from the [SampleCache](#), so no file is read again.

The renderer owns its own instruments and its own copy of every note. It is a snapshot, like the copies in the [undo history](#): the export goes on with the song as it was, whatever you do in the window meanwhile.

How long is a song?

The song's length is known in steps; the file's length is in frames. `played_seconds` adds up each step at the tempo in force there, so a song with tempo changes, ritardandos and fermatas still gets the right length. Then comes a **tail** of two seconds, to let the last notes ring out. With the reverb on, the tail also waits for its decay and pre-delay, so the room can die away instead of being cut off.

The starting song is 128 steps at 130 BPM. There are 4 steps to a beat, so a step lasts $60 / 130 / 4 = 0.115$ seconds, 5,088.46 frames at 44,100 (a step is not a whole number of frames; the [sequencer](#) keeps the fraction). The 128 steps take 651,323 frames; with a two-second tail of 88,200 the file has 739,523 frames, 16.77 seconds.

Silent steps after the end

There is a catch. In song mode the audio thread *loops*: when the playhead reaches the end of the song, it starts again from bar 1. The tail would end with the first chord of the song starting again. The fix is to tell this renderer, and only this one, that the song is longer than it really is, by some silent steps. For the starting song that is $88,200 / 5,088.46 = 17.33$, rounded up to 18, plus one: 19 silent steps.

Why the extra one? A step is not a whole number of frames, so the sequencer starts some steps a frame earlier or later than the exact fraction. The extra step means the end of the file is always a whole step away from the loop, whatever the rounding.

The silent steps play at the tempo the song ends at. The audio thread plays steps past the end of the tempo list at its last tempo, so a song that ends with a *ritardando* gets a tail measured in its slow final steps, not in the faster ones of the start.

Writing a WAV file

A WAV file is a short header (44 bytes here) saying how the sound is stored, followed by the samples. The export writes the most common kind:

- **44,100 frames per second**, the rate of a CD, which every program understands,
- **2 channels** (stereo), stored **interleaved**: left, right, left, right..., exactly like the buffers `cpal` hands the audio thread,
- **16 bits per sample**: each sample is an `i16`, a whole number from -32,768 to 32,767.

That makes $44,100 \times 2 \times 2 = 176,400$ bytes per second of sound, about 10.6 MB per minute. The starting song's file is $44 + 739,523 \times 4 = 2,958,136$ bytes.

Writing the header and the samples by hand would be easy, but the `hound` crate already does it, and the sampler uses it to *read* WAV files anyway:

```

pub fn export_wav_cancellable(
    renderer: &mut Renderer,
    path: &Path,
    frames: usize,
    mut progress: impl FnMut(usize),
    cancel: &AtomicBool,
) -> Result<WavExport, hound::Error> {
    let spec = hound::WavSpec {
        channels: renderer.channels() as u16,
        sample_rate: renderer.sample_rate(),
        bits_per_sample: 16,
        sample_format: hound::SampleFormat::Int,
    };
    let mut temporary = PathBuf::from(path);
    temporary.as_mut_os_string().push(".tmp");

    let mut writer = hound::WavWriter::create(&temporary, spec)?;
    let mut block = vec![0.0_f32; BLOCK_FRAMES * renderer.channels()];
    let mut done = 0;
    while done < frames {
        if cancel.load(Ordering::Relaxed) {
            drop(writer);
            let _ = fs::remove_file(&temporary);
            return Ok(WavExport::Cancelled);
        }
        let count = (frames - done).min(BLOCK_FRAMES);
        let samples = &mut block[..count * renderer.channels()];
        renderer.render(samples);
        for &sample in samples.iter() {
            writer.write_sample(to_i16(sample))?;
        }
        done += count;
        progress(done);
    }
    writer.finalize()?;
    fs::rename(&temporary, path)?;
    Ok(WavExport::Written)
}

```

The sound is made in blocks of 4,096 frames (`BLOCK_FRAMES`), in one buffer reused for every block. The last block is usually shorter, so `samples` is a slice of just the part needed. After each block, `progress` is called with the number of frames done. Its type is `impl FnMut(usize)` : any closure that takes a number and may change what it captured. The app's closure stores the number where the status bar can read it.

Every `?` here returns a `hound::Error`. `fs::rename` returns a `std::io::Error` instead, and `?` still works, because `hound::Error` implements `From<std::io::Error>`: the `?` operator converts the error on the way out.

`export_wav` is a thin wrapper for callers that cannot cancel. It passes a flag that is never set and returns a plain `Result<(), hound::Error>`; the tests use it. The app uses the cancellable version.

Write to a temporary file, then rename

The samples go to `song.wav.tmp` first, and only when everything is written is it renamed to `song.wav`. This is how the project file is [saved](#) too. Exporting over an older `song.wav` leaves the old file whole until the new one is complete: if the disk fills up or something fails halfway, you are left with your old export, not half of a new one.

`as_mut_os_string().push(".tmp")` adds to the name. `set_extension("tmp")` would *replace* `.wav`, giving `song.tmp`, which could be the name of someone else's file.

From f32 to i16

```
/// Turns a sample between -1.0 and 1.0 into a 16-bit integer sample.
fn to_i16(sample: f32) -> i16 {
    (sample.clamp(-1.0, 1.0) * f32::from(i16::MAX)).round() as i16
}
```

`clamp` keeps the sample between -1.0 and 1.0, then it is scaled to -32,767..32,767 and rounded to the nearest whole number. An `as` cast from a float to an integer already stops at the integer's limits in Rust (it *saturates*: `3e9 as i16` is 32,767), so the clamp is not needed for safety, but it says what is meant. `f32::from(i16::MAX)` is a conversion that can never lose anything, so it is a `From`, not an `as`.

The audio thread already keeps the master output inside -1.0..1.0 with its [soft clipper](#) (`tanh`), so in practice the clamp never cuts anything.

A thread of its own

Rendering takes a while. With the built-in sounds, a release build on a four-core computer plays Boléro (fourteen minutes, for a full orchestra) through a renderer in about twenty seconds, some forty times faster than it plays; the starting song takes a fraction of a second. A debug build is many times slower. The window must not freeze meanwhile, so the export runs on a **thread** of its own:

```

let done = Arc::new(AtomicUsize::new(0));
let cancel = Arc::new(AtomicBool::new(false));
// The thread gets its own `Arc`s pointing to the same counter and
// flag, and its own copy of the path. `move` gives the closure
// ownership of them, and of the renderer: from now on they belong
// to the thread.
let worker = {
    let done = Arc::clone(&done);
    let cancel = Arc::clone(&cancel);
    let path = path.clone();
    thread::spawn(move || {
        match export_wav_cancellable(
            &mut renderer,
            &path,
            frames,
            |frames_done| done.store(frames_done, Ordering::Relaxed),
            &cancel,
        ) {
            Ok(WavExport::Written) => Ok(()),
            Ok(WavExport::Cancelled) => Err(CANCELLED.to_string()),
            Err(err) => Err(err.to_string()),
        }
    })
};
self.export = Some(Export::new(path, frames, done, Some(cancel), worker));

```

`std::thread::spawn` starts a new thread running a closure, and asks two things of it, the same two cpal asks of the [audio callback](#):

- **'static**: the closure may not borrow anything from the code that started it. The new thread can outlive the function that spawned it (it certainly outlives this frame of the UI), so a borrowed reference could point at something that is gone. That is why the closure is marked **move**: it takes *ownership* of everything it uses (the renderer, the path, the counter, the flag) instead of borrowing it.
- **Send**: everything moved into it must be safe to hand to another thread. The renderer qualifies, because everything in it does: the instruments, the `Arc`s with the samples, the ring buffer ends. If it held an `Rc` (a reference count that is not thread-safe), this line would not compile.

The block `{ let done = Arc::clone(&done); ... }` is a common trick. Both the UI and the thread need the counter, so each gets its own `Arc` pointing at it. Inside the block, the new `done` *shadows* the outer one, and the **move** closure takes the inner one, leaving the outer `done` for the UI. The same goes for `cancel` and `path`.

Progress: an atomic counter

The thread writes how many frames are done into an `AtomicUsize`, and the status bar reads it when it draws the progress bar, as the window reads the `playhead`.

`Ordering::Relaxed` is enough: no other data hangs on the number, and a bar one frame behind is fine. The file is only trusted once the thread has finished, and `join` (below) gives the full guarantee.

Knowing when it is done

`spawn` returns a `JoinHandle`. The app keeps it, with the counter, in an `Export`:

```
pub(super) struct Export {
    path: PathBuf,
    /// How much there is to do: frames of sound, or pictures.
    frames: usize,
    /// How many are done. The export thread writes it and the UI
    /// thread reads it; an atomic needs no lock for that.
    done: Arc<AtomicUsize>,
    /// Set to ask the thread to stop, for exports that can be cancelled.
    cancel: Option<Arc<AtomicBool>>,
    /// The thread doing the work. Joining it gives back what its closure
    /// returned: here, whether the file was written.
    worker: JoinHandle<Result<(), String>>,
}
```

The video export uses the same struct, counting pictures instead of frames.

`JoinHandle<Result<(), String>>` says what the thread's closure returns: `Ok()` if the file was written, or a message. The error is turned into a `String` on the export thread (`err.to_string()`), since a message for the status bar is all the window needs. A stopped export also comes back as an `Err`, with the special string `CANCELLED`.

Every frame, the app checks on it:

```

pub(super) fn check_export(&mut self, ctx: &egui::Context) {
    let Some(export) = &self.export else {
        return;
    };
    if !export.worker.is_finished() {
        ctx.request_repaint_after(Duration::from_millis(100));
        return;
    }
    // `take` moves the export out of the `Option` and leaves `None`,
    // so `join` can take ownership of the thread's handle.
    if let Some(export) = self.export.take() {
        let name = export.file_name();
        match export.worker.join() {
            Ok(Ok(())) => self.notice = Some(trf("Exported {}", &
[&name])),
            Ok(Err(err)) if err == CANCELLED => {
                self.notice = Some(trf("Stopped exporting {}", &[&name]));
            }
            Ok(Err(err)) => {
                self.message = Some(trf("Could not export {}: {}", &
[&name, &err]));
            }
            // `join` fails only if the thread panicked.
            Err(_) => self.message = Some(trf("Exporting {} failed", &
[&name])),
        }
    }
    // ...and closes the window, if a quit was waiting for the export.
}

```

A few things to notice:

- `egui` only draws a new frame when something happens: a mouse move, a key. While an export runs, nothing might, and the progress bar would stand still. `request_repaint_after(100 ms)` asks for another frame soon, ten times a second, until the export is done.
- `is_finished()` asks without waiting. `join()` *waits* for the thread to end, but it is only called once `is_finished()` is true, so it returns at once.
- `join` takes the handle by value: it consumes it, since a thread can only be joined once. The handle is inside `self.export`, an `Option`, and you cannot move something out of a field you only borrow. `Option::take` moves the value out and leaves `None` in its place, so the export is gone from `self` and owned by the local `export`.
- The `match` has two layers: `join` returns a `Result` that is `Err` only if the thread panicked, and inside its `Ok` is the closure's own `Result`. Patterns nest, so `Ok(Ok(()))` matches “the thread ended normally, and the file was written”. The arm with `if err == CANCELLED` is a *match guard*: it only matches when the condition holds, and the next arm takes the other errors.

Stopping an export

While an export runs, the status bar shows a progress bar, **Exporting song.wav**, and a small **Stop** button:

```
if let Some(export) = &self.export {
    ui.separator();
    ui.add(
        egui::ProgressBar::new(export.progress())
            .desired_width(160.0)
            .text(trf("Exporting {} ", &[&export.file_name()])),
    );
    if export.can_cancel() && ui.small_button(tr("Stop")).clicked() {
        export.cancel();
    }
}
```

`cancel` stores `true` in the `AtomicBool`, and the export thread checks it before each block, as `export_wav_cancellable` above shows. A stopped export drops the writer, deletes its temporary file and returns `WavExport::Cancelled`, so a half-written WAV is never left where a finished one should be. The writer is dropped first, which closes the file: Windows does not delete a file that is still open.

Closing the window while an export runs asks first: **An export is still running**, with **Stop it and quit** and **Keep exporting**. It works like the [question about unsaved changes](#), which comes first if there are any: the app cancels the close with `ViewportCommand::CancelClose`, and a modal asks. Choosing **Stop it and quit** sets the same flag and remembers that a close is waiting; when `check_export` sees the thread finish, it closes the window. Waiting for the thread gives it time to remove its temporary file before the process ends.

MIDI files: written at once

A MIDI file holds only the notes, a few bytes each. The whole of a long song is a few kilobytes, made in a moment, so **Export song as MIDI...** and **Export pattern as MIDI...** need no thread and no progress bar: they ask where to save and write the bytes with `std::fs::write`. How `song_to_midi` and `pattern_to_midi` turn notes into those bytes is in [MIDI](#).

Every export offers a file name made from the project's: `FileState::export_name` turns `song.lineup` into `song.wav`, `song.mp4` or `song.mid`, or suggests `Untitled.wav` for a project that was never saved. A pattern's MIDI file is named after

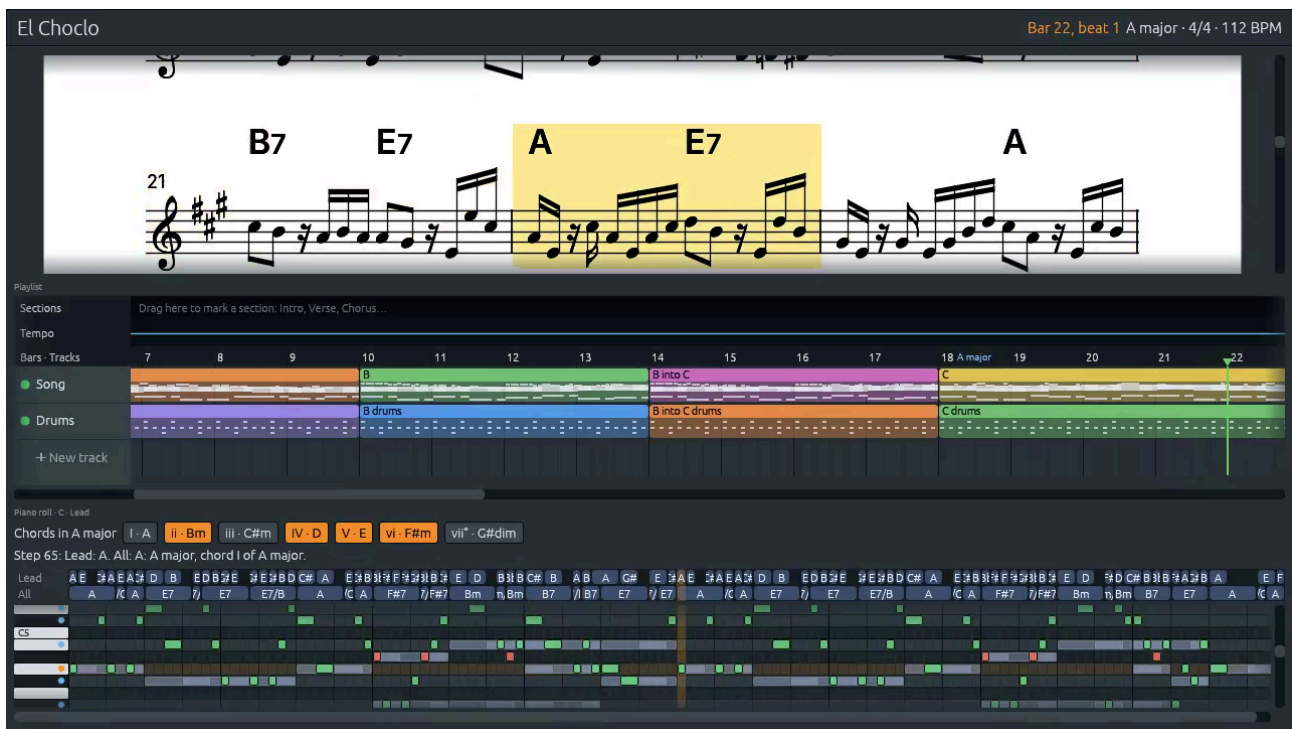
the pattern, with any `/` or `\` replaced, since a file name cannot hold them. If you type a name without an extension, the right one is added.

A video: the song with the views playing along

A WAV file carries the song, but not the harmony you see while it plays. **Export video...** saves an MP4 file with the sound and the views you pick following the music: the bar playing framed in the chord sheet, the chord lit in the harmony keys, the playhead moving across the playlist and the piano roll, and each key change shown as it comes.

The screenshot displays the 'El Choclo' interface, which is a music notation and harmony tool. At the top, the title 'El Choclo' is shown on the left, and 'Bar 29, beat 1 A harmonic minor · 4/4 · 112 BPM' is on the right. Below the title is a 'Chord sheet' section containing a grid of 32 measures. Each measure contains a chord symbol and its Roman numeral. The grid is organized into four columns and eight rows. The first column contains measures 1, 5, 9, 13, 17, 21, 25, and 29. The second column contains measures 2, 6, 10, 14, 18, 22, 26, and 30. The third column contains measures 3, 7, 11, 15, 19, 23, 27, and 31. The fourth column contains measures 4, 8, 12, 16, 20, 24, 28, and 32. The interface also features a 'Key' section at the bottom left with a dropdown menu set to 'A' and a 'harmonic minor' button. Below this is a 'Time' section with a dropdown set to '4/4' and a '16 steps a bar' button. To the right of these are buttons for 'Triads', '7ths', and '9ths'. At the bottom left, there is a 'Short' dropdown set to 'E7 → Am: back home' and a search bar labeled 'Find the key'. On the right side of the bottom section is a piano keyboard diagram with keys colored to match the harmony keys.

A dialog asks which views to show, and how big the pictures are: **1280 × 720** (HD) or **1920 × 1080** (Full HD). Its preview shows a picture of the video at the bar you pick with **Preview at**. On it, each view's box can be dragged, resized by its corner and zoomed by scrolling over it, from 50% to 250%. **Stack** puts the views back one above the other.



Making a video takes four pieces of work, and the next sections take them one at a time:

1. **Draw** the views as pictures, without a window or a graphics card.
2. **Compress** the pictures as H.264 and the sound as MP3.
3. **Put both** in an MP4 file.
4. **Keep the picture with the sound**, tempo changes and all.

Nothing to install

The usual way to make a video from a program is to hand pictures to FFmpeg. But FFmpeg is a separate program that Windows and macOS do not have, so every user would have to install it. Instead, the app builds two well-known encoders from their C sources along with itself, through two crates:

What	Format	Encoder	Crate
Pictures	H.264, 30 a second	Cisco's OpenH264 (BSD licence)	<code>openh264</code>
Sound	MP3, 256 kbit/s stereo	LAME (LGPL)	<code>mp3lame-encoder</code>
Both together	MP4	our own, a few hundred lines	<code>(lineup-video)</code>

A crate with C inside has a `build.rs` that runs a C compiler before Rust compiles the rest, using the `cc` crate. The `Cl` builds on Linux, Windows and both kinds of Mac, so it

proves this works everywhere. Both encoders, and the crate that paints the pictures, are optimized even in debug builds, like the SoundFont synthesizer, for the same reason given in [Debug builds](#).

H.264 and MP3 are not the newest formats. They were picked because every player, phone, browser and video site reads them.

The crate: `lineup-video`

The crate knows nothing about songs. It takes pictures and sound and writes a file. Its only dependencies are the two encoders and `epaint`, the part of `egui` that describes shapes. It has four modules:

- `canvas` paints `egui`'s shapes into a picture in memory.
- `h264` compresses each picture.
- `mp3` compresses the sound.
- `mp4` writes the file.

and one type that uses them all, `VideoWriter`. You give it a picture, then the sound that plays while it is shown, and so on to the end:

```
let mut writer = VideoWriter::create(&path, settings)?;
```

```
match scene.draw(time, position) {  
    Some(picture) => writer.add_picture(picture)?,  
    None => writer.repeat_picture(),  
}  
renderer.render(&mut sound);  
writer.add_sound(&sound)?;
```

```
match record() {  
    Ok(()) => writer.finish(),  
    Err(err) => {  
        writer.cancel();  
        Err(err)  
    }  
}
```

`create` writes to `name.mp4.tmp`, `finish` renames it, and `cancel` removes it, as for a WAV file. `cancel(self)` takes the writer by value, so a cancelled writer cannot be used again: the compiler stops you.

Inside an MP4 file

An MP4 file is a tree of **boxes**. Each box is a 4-byte length, a 4-letter name, and its contents, which may be more boxes:

```
ftyp          "this is an MP4 file"
mdat          all the compressed pictures and sound, one after the
other
moov          the index:
  mvhd        how long the whole thing lasts
  trak        the pictures:
    tkhd, mdia ... size, clock, and the tables below
    avc1 + avcC  "H.264", and the encoder's parameter sets
    stts        how long each picture is shown
    stss        which pictures are key pictures
    stsz, stco   how big each one is, and where it is in mdat
  trak        the sound:
    edts/elst   where the sound really starts (see below)
    mp4a + esds "MP3"
    stts, stsz, stco the same tables for the MP3 frames
```

A picture or a piece of sound in `mdat` is a *sample*, and the tables in `moov` list every sample of a track. The format is ISO/IEC 14496-12 and 14496-14; only what a video with one H.264 track and one MP3 track needs is written.

A box whose size comes last

A box starts with its size, but the size is only known once its contents are written.

`write_box` leaves room for it, lets a closure write the contents, and then fills it in:

```
/// Writes a box: its size, its name, then whatever `contents` writes.
/// The size is only known afterwards, so it is filled in at the end.
fn write_box(out: &mut Vec<u8>, name: &[u8; 4], contents: impl FnOnce(&mut
Vec<u8>)) {
    let start = out.len();
    out.extend_from_slice(&[0; 4]);
    out.extend_from_slice(name);
    contents(out);
    let size = (out.len() - start) as u32;
    out[start..start + 4].copy_from_slice(&size.to_be_bytes());
}
```

Boxes inside boxes are then closures inside closures, and the code has the shape of the tree:

```

write_box(out, b"trak", |out| {
    write_tkhd(out, 1, duration_ms, false, format.width, format.height);
    write_box(out, b"mdia", |out| {
        write_mdhd(out, format.timescale, self.video.duration());
        write_hdlr(out, b"vide", "Pictures");
    });
});

```

`b"trak"` is a *byte string*: a `&[u8; 4]`, four bytes, not a `&str`. The parameter's type `&[u8; 4]` makes the compiler check that every name is exactly four letters. `to_be_bytes` gives a number's bytes *big-endian*, the most significant first, which is how MP4 (and MIDI) store numbers.

The index comes last

`moov` has to come after the data. Until the end, nobody knows how big each picture is or where it landed. So `Mp4Writer` writes `mdat` as it goes, keeps a list of sizes and offsets, and writes `moov` in `finish`. Then it goes back to fill in `mdat`'s length:

```

// A size of 1 says the real size follows as 64 bits, so the file
// can grow past 4 GB. It is filled in at the end.
head.extend_from_slice(&1_u32.to_be_bytes());
head.extend_from_slice(b"mdat");
head.extend_from_slice(&0_u64.to_be_bytes());

```

```

self.file.seek(SeekFrom::Start(self.mdat_start + 8))?;
self.file
    .write_all(&(self.position - self.mdat_start).to_be_bytes())?;
self.file.seek(SeekFrom::Start(self.position))?;

```

A long Full HD video can pass 4 GB, so the size is 64 bits. For the same reason the table of where each sample starts is `stco` (32-bit offsets, which every player reads) while the file is small enough, and `co64` once an offset passes `u32::MAX`.

How long each picture is shown

`stts` stores *runs*: "300 pictures of 1/30 s each" takes one entry, not 300:

```
// How long the samples last, as runs of samples of the same length.
let mut runs: Vec<u32, u32> = Vec::new();
for &duration in &track.durations {
    match runs.last_mut() {
        Some((count, last)) if *last == duration => *count += 1,
        _ => runs.push((1, duration)),
    }
}
```

`last_mut` gives a mutable reference to the last run, if there is one. The match guard `if *last == duration` extends it when the new duration is the same; otherwise a new run starts.

The pictures' clock runs at 90,000 ticks a second, the usual one for video, which divides evenly into 24, 25, 30 or 60 pictures a second. At 30 a second, a picture lasts 3,000 ticks. When nothing on screen changes, there is no new picture at all:

`repeat_picture` makes the last one last 3,000 ticks longer. A screen where only a frame moves once a bar costs almost nothing.

Making the sound start on time

An MP3 encoder puts 576 samples of silence before the sound, and a decoder adds 529 more. Without a fix, the music would be 1,105 samples, 25 ms, late. That is almost one picture, and enough to see a chord light up before you hear it. The sound track's **edit list** (`elst`) tells the player where the real sound starts:

```
// The edit list: play the track from `priming` on, for as long
// as the real sound lasts. The silence the encoder put first,
// and the padding after the last frame, are skipped.
write_box(out, b"edts", |out| {
    write_full_box(out, b"elst", 0, 0, |out| {
        out.extend_from_slice(&1_u32.to_be_bytes());
        out.extend_from_slice(&(duration_ms as u32).to_be_bytes());
        out.extend_from_slice(&format.priming.to_be_bytes());
        out.extend_from_slice(&1_u16.to_be_bytes()); // at normal speed
        out.extend_from_slice(&0_u16.to_be_bytes());
    });
});
```

One edit: play for as long as the real sound lasts, starting 1,105 samples into the track, at normal speed. To check it, a test video was made with clicks at 1.0, 2.5 and 4.0 s and a white flash on the pictures at the same moments. Decoded with FFmpeg, the clicks are at exactly 1.0, 2.5 and 4.0 s, and the flashes on pictures 30, 75 and 120.

Pictures: H.264

The pictures arrive as RGBA, four bytes a pixel. H.264 wants them as **YUV**: brightness (Y) for every pixel, and two colour differences (U and V) for every square of four pixels. The eye sees detail in brightness much better than in colour, so that alone halves the size. After the first picture, the encoder only stores what changed. Every two seconds there is a *key picture* that stands on its own, so a player can jump there and start drawing at once.

The encoder is set up for what a video of the app shows: a screen, with sharp edges, flat colours and text:

```
let config = EncoderConfig::new()
// A screen recording: sharp edges, flat colours, text.
.usage_type(UsageType::ScreenContentRealTime)
.profile(Profile::Baseline)
.rate_control_mode(RateControlMode::Bitrate)
.bitrate(BitRate::from_bps(bitrate(width, height)))
.max_frame_rate(FrameRate::from_hz(frame_rate as f32))
// Every picture is kept: dropping one to save room would
// make the playhead stutter.
.skip_frames(false)
.complexity(Complexity::High)
// A key picture every two seconds, so a player can jump.
.intra_frame_period(IntraFramePeriod::from_num_frames(frame_rate * 2))

.vui(VuiConfig::bt709().matrix_coefficients(MatrixCoefficients::Smpte170M)
);
```

The bit rate is 4.5 bits a pixel a second: about 4 Mbit/s at 1280×720 and 9 at 1920×1080, which keeps text sharp and the file small.

What comes out is a list of *NAL units*: two parameter sets that describe the stream (the picture size, the profile), then the pictures. The encoder puts `0 0 1` or `0 0 0 1` before each unit to mark where it starts. An MP4 file wants each unit after its length instead, so the marks come off, with the [slice patterns](#) MIDI messages are read with:

```
/// A NAL unit without the 0 0 1 or 0 0 0 1 the encoder puts before it to
/// mark where it starts.
fn without_start_code(nal: &[u8]) -> &[u8] {
    match nal {
        [0, 0, 0, 1, rest @ ..] | [0, 0, 1, rest @ ..] => rest,
        _ => nal,
    }
}
```


`rest @ ..` binds the rest of the slice to a name, and `|` joins two patterns in one arm, since both bind `rest`. The result borrows from the argument, so nothing is copied.

The kind of each unit is in the low five bits of its first byte. The parameter sets are kept aside, because the MP4 file stores them once, in its `avcC` box; the rest are the picture:

```
match header & 0x1F {
    SEQUENCE_PARAMETERS => self.sps = Some(nal.to_vec()),
    PICTURE_PARAMETERS => self.pps = Some(nal.to_vec()),
    kind => {
        picture.key |= kind == KEY_PICTURE;
        picture
            .bytes
            .extend_from_slice(&(nal.len() as u32).to_be_bytes());
        picture.bytes.extend_from_slice(nal);
    }
}
```

A `match` arm can compare against constants (`SEQUENCE_PARAMETERS` is `7`), and the last arm, a plain name, catches every other value and binds it to `kind`.

Sound: MP3

LAME takes the samples and gives back MP3 *frames*: each holds 1,152 samples of every channel and starts with a 4-byte header. The MP4 file keeps each frame as one sample of the sound track, so the frames are cut apart as they come, by reading each one's length from its header:

```
fn frame_length(bytes: &[u8]) -> Option<usize> {
    let [first, second, third, ..] = *bytes else {
        return None;
    };
    // Sync, MPEG-1 (11), Layer III (01).
    if first != 0xFF || second & 0xFE != 0xFA {
        return None;
    }
    const KBPS: [usize; 15] = [
        0, 32, 40, 48, 56, 64, 80, 96, 112, 128, 160, 192, 224, 256, 320,
    ];
    const RATES: [usize; 3] = [44_100, 48_000, 32_000];
    let kbps = *KBPS.get(usize::from(third >> 4))?;
    let rate = *RATES.get(usize::from((third >> 2) & 0b11))?;
    let padding = usize::from((third >> 1) & 1);
    (kbps > 0).then(|| 144 * kbps * 1000 / rate + padding)
}
```

The header's first eleven bits are all set, so a decoder can find a frame; then come the version and the layer, and in the third byte the bit rate and the sample rate, as positions in a table. At 256 kbit/s and 44.1 kHz a frame is $144 \times 256,000 / 44,100 = 835.9$ bytes: 835, or 836 when the padding bit is set, so that on average the frames come out at exactly the bit rate.

`let [first, second, third, ..] = *bytes else { return None; }` is a *let-else*: it takes the first three bytes if there are at least three, and returns otherwise. `KBPS.get(...)?` returns `None` for a number past the end of the table, instead of panicking as `KBPS[...]` would: the bytes could be anything.

The only unsafe in the program

LAME is C, and it writes its output into memory we give it. The crate wants that memory as `&mut [MaybeUninit<u8>]`: bytes that may hold anything yet. A `Vec` has room past its end once it has reserved some, and `spare_capacity_mut` hands that room over:

```
let frames = samples.len() / 2;
self.pending
    .reserve(mp3lame_encoder::max_required_buffer_size(frames));
let written = self
    .encoder
    .encode(InterleavedPcm(samples), spare(&mut self.pending))
    .map_err(|err| err.to_string())?;
// SAFETY: LAME wrote `written` bytes into the spare room.
unsafe { self.pending.set_len(self.pending.len() + written) };
Ok(self.cut())
```

Then the `Vec` has to be told that it is longer now. `set_len` is `unsafe`: Rust cannot check that the bytes it adds to the `Vec` were really written, and reading bytes that were not is undefined behaviour. An `unsafe` block is where the programmer takes over a promise the compiler cannot check, and the `// SAFETY:` comment says why the promise holds: LAME said it wrote `written` bytes there. The two blocks in `mp3.rs` (here and in `finish`) are the only `unsafe` code in the whole program.

The other way would be a buffer of zeros, written over by LAME. That is safe and a little slower, since every byte is written twice. Either is fine here; this one shows what `unsafe` is for.

Drawing the app without a window

The views in a video are ordinary egui code. The chord sheet in the video *is* the chord sheet in the app, through small `video` functions that call the same drawing code without the menus. But egui does not draw pixels itself. Each frame, it turns the UI into a list of **triangles**, each corner with a colour and a point in a texture (the texture holds the font's letters). In the app, `egui_glow` sends those triangles to the graphics card.

The export cannot count on a graphics card, and it runs on a thread of its own. So each view gets a **pane**: an `egui::Context` of its own, with no window, and a `Canvas` that paints the triangles on the processor. Each picture of the video, a pane runs the view's code as `eframe` runs the window's, tessellates what it drew into triangles, and paints them:

```
let ctx = self.ctx.clone();
let mut output = ctx.run_ui(input, &mut contents);
let shapes = std::mem::take(&mut output.shapes);
let primitives = ctx.tessellate(shapes, output.pixels_per_point);
changed |= self.canvas.paint(
    primitives,
    &mut output.textures_delta,
    output.pixels_per_point,
    background,
);
```

`textures_delta` is what egui added to its textures this frame: the font picture grows as new letters and sizes are needed, and the canvas keeps its own copy, as a graphics card would.

Painting a triangle

To paint a triangle, visit each pixel near it and ask “is this pixel inside?”. The test is the **edge function**:

```
/// Twice the area of the triangle `from`, `to`, `p`: positive when `p`
/// is on the inside of the edge from `from` to `to`, zero on it.
fn edge(from: (i64, i64), to: (i64, i64), p: (i64, i64)) -> i64 {
    (to.0 - from.0) * (p.1 - from.1) - (to.1 - from.1) * (p.0 - from.0)
}
```

For an edge from A to B, this cross product is positive on one side and negative on the other. A pixel is inside when all three edges agree. The same three numbers, divided by their total, say how close the pixel is to each corner. That blends the corners'

colours and texture coordinates, which is how letters get their shapes. A few details make it match the screen:

- **Whole numbers, in 1/256 of a pixel.** Two triangles that share an edge must not both paint the pixels on it, or a half-transparent shape gets a darker seam. With whole numbers there is no rounding doubt, and the *top-left rule* settles the ties: a pixel exactly on an edge belongs to the triangle only if that edge is a top or a left one.
- **Premultiplied colours.** egui stores colours already multiplied by their opacity, and blends with `new + old × (1 - opacity)`, as its graphics-card shader does.
- **Stepping, not multiplying.** Moving one pixel right changes each edge function by a constant, so the inner loop adds instead of multiplying. For plain one-colour triangles, most of a UI, a fast path skips the texture altogether.

Bands of rows, on every core

The picture is cut into bands of rows, and each band is painted on its own core:

```
self.pixels.fill(background.to_array());
let width = self.width;
let cores = thread::available_parallelism().map_or(1, |n| n.get().min(8));
let rows_per_band = self.height.div_ceil(cores).max(1);
let textures = &self.textures;
thread::scope(|scope| {
    for (band, pixels) in self.pixels.chunks_mut(rows_per_band *
width).enumerate() {
        let top = band * rows_per_band;
        let bottom = top + pixels.len() / width;
        scope.spawn(move || {
            let mut band = Band {
                pixels,
                width,
                top,
                bottom,
            };
            for clipped in primitives {
                if let Primitive::Mesh(mesh) = &clipped.primitive {
                    band.paint_mesh(mesh, clipped.clip_rect,
pixels_per_point, textures);
                }
            }
        });
    }
});
```

These threads *borrow*: each gets a `&mut` slice of the picture, and all of them share `&self.textures` and the triangles. `thread::spawn` could not do that, since its closure

must be `'static . std::thread::scope` can, because it does not return until every thread spawned in it has finished, so the borrows cannot outlive what they point at. The compiler checks the rest: `chunks_mut` gives slices that do not overlap, so no two threads can write the same pixel.

Each band paints every triangle, but only its own rows of it. A triangle that crosses two bands is painted half by each.

A box for each view, and zoom

Zoom works the way egui's own zoom does. The pane tells its context that there are more pixels to a point, 1.4 times as many at 140%, so egui lays the view out in fewer points and draws every letter at the bigger size. If the view were drawn small and the picture stretched instead, the letters would come out blurred:

```
let ppp = self.pixels_per_point;
let size = vec2(self.canvas.width() as f32, self.canvas.height() as f32) /
ppp;
let mut input = egui::RawInput {
    screen_rect: Some(Rect::from_min_size(Pos2::ZERO, size)),
    time: Some(time),
    predicted_dt: 1.0 / frame_rate as f32,
    ..Default::default()
};
input
    .viewports
    .entry(egui::ViewportId::ROOT)
    .or_default()
    .native_pixels_per_point = Some(ppp);
```

`RawInput` is what eframe gives egui each frame: the screen's size, the time, the mouse and keys. Here it is made up, with no mouse and no keys, and the *video's* time, so anything that moves with time moves with the video. `..Default::default()` fills in every field not named.

Full HD lays the views out exactly like HD, 1280 by 720 points, at 1.5 pixels a point: the text is sharper, but nothing more fits. Each pane paints its own canvas, and `Canvas::copy_from` copies the panes into the picture, in their boxes, under a title line with the song's name, the bar and beat, the key, the time signature and the tempo. Boxes may overlap: the view lower in the list is copied last, on top.

On the beat

Two details keep the views on the beat:

- **No gliding.** In the window, a view scrolls smoothly to the bar playing, taking about half a second. In a video that is half a second late, so each pane turns egui's scroll animation off and jumps there: `style.scroll_animation = egui::style::ScrollAnimation::none()`.
- **Not a picture late.** A view that scrolls only moves on the *next* frame, and asks egui for one at once (a repaint delay of zero). The pane runs a second pass within the same picture when that happens, so the bar shows on its first beat. A pane whose size or zoom just changed is always drawn twice too, because egui measures some things on its first frame.

Pictures that did not change

When a song is playing, most pictures look like the one before: the chord sheet only moves when the bar changes. `Canvas::paint` keeps the last picture's triangles, and paints only if the new ones differ:

```
/// Whether two lists of shapes paint the same picture.
fn same(new: &[ClippedPrimitive], old: &[ClippedPrimitive]) -> bool {
    new.len() == old.len()
    && new.iter().zip(old).all(|(new, old)| {
        new.clip_rect == old.clip_rect
        && match (&new.primitive, &old.primitive) {
            (Primitive::Mesh(new), Primitive::Mesh(old)) => new ==
old,
            _ => false,
        }
    })
}
```

Comparing two meshes with `==` compares every corner of every triangle, which is far cheaper than painting them. When no pane changed and no box moved, `Scene::draw` returns `None`, the encoder does not run, and `repeat_picture` makes the last picture last longer. Skipping unchanged pictures and painting plain triangles on the fast path make a video export several times faster than painting every picture in full.

Settings that live on a thread

Note names, translations and the minor-key numerals are [per-thread settings](#), `thread_local!` values that the window sets before drawing each frame. The export draws on a thread of its own, which starts with the defaults: a Spanish user's video would come out in English, in letters. So the export reads the three settings on the UI

thread, moves them into the closure (they are small `Copy` values), and sets them first thing on its own thread:

```
let numerals = MinorNumerals::current();
let language = crate::i18n::Language::current();
let naming = lineup_core::NoteNaming::current();
```

```
thread::spawn(move || {
    // The views are made here, on the export's own thread,
    // and write numerals, words and note names as the app does.
    numerals.set();
    language.set();
    naming.set();
    // ...then the scene, the writer and the pictures, below.
})
```

Keeping the picture with the sound

The song is played by the same off-line `Renderer` as a WAV export, 1/30 s at a time: at 44,100 samples a second, exactly 1,470 samples a picture. The pictures must show *where the music is*, and with tempo changes, ritardandos and fermatas that is not a simple sum. So the renderer reports it. After each block of sound, the audio thread stores its position in steps, with a fraction (4.5 is halfway through step 4), as it does for the window, and `Renderer::position` returns it:

```
for picture in 0..pictures {
    if cancel.load(Ordering::Relaxed) {
        return Err(CANCELLED.to_string());
    }
    let time = picture as f64 / f64::from(FRAME_RATE);
    // Before the first sound, the song is about to
    // start; after it, where the sound so far ended.
    let position = if picture == 0 {
        Some(0.0)
    } else {
        renderer.position()
    };
    match scene.draw(time, position) {
        Some(picture) => writer.add_picture(picture)?,
        None => writer.repeat_picture(),
    }
    renderer.render(&mut sound);
    writer.add_sound(&sound)?;
    done.store(picture + 1, Ordering::Relaxed);
}
```

Each picture is drawn *before* the sound that plays while it is shown, at the position where that sound starts.

The position counts steps *as played*: with a repeat, or a chorus the band plays three times, that is not the place in the playlist. `Scene::draw` maps it back with the song's arrangement, as the window does for its playhead:

```
let at = position.and_then(|position| {
    let step = position.max(0.0) as usize;
    let pass = *self.song.pass_at(step)?;
    Some((pass.from + (step - pass.start), pass))
});
```

A *pass* is one stretch of the playlist played once: from where in the playlist (`from`), starting at which played step (`start`). How the arrangement unrolls repeats into passes is in [Song form](#).

To check, El Choclo was exported. At 112 BPM in 4/4, bar 15 starts at 30.0 s, bar 29 at 60.0 s and bar 43 at 90.0 s, and the [playback trace](#), which plays through the same renderer, has the first notes of those bars at exactly those times. The video's pictures there show "Bar 15, beat 1", "Bar 29, beat 1" and "Bar 43, beat 1".

Arranging the views

Where each box goes and how big its view is drawn is a `Layout`: one `Placement` for each of the five views.

```
/// Where a view goes in the picture, and how big it is drawn.
#[derive(Debug, Clone, Copy, PartialEq, Serialize, Deserialize)]
#[serde(default)]
pub struct Placement {
    pub shown: bool,
    /// Its box: left, top, right and bottom, from 0 to 1 across the room
    /// under the title line.
    pub rect: [f32; 4],
    /// How big it is drawn: 1 as in the window, 2 twice as big.
    pub zoom: f32,
}
```

The box is kept in parts of the room, from 0 to 1, not in pixels, so the same layout fits both sizes, and the preview in the dialog, which is 640 points wide.

The placements live in an array, but the code reads them by view:

`layout[View::ChordSheet]`. That works because `Layout` implements the indexing

traits, with a `View` as the index:

```
impl std::ops::Index<View> for Layout {
    type Output = Placement;
    fn index(&self, view: View) -> &Placement {
        &self.views[view.index()]
    }
}

impl std::ops::IndexMut<View> for Layout {
    fn index_mut(&mut self, view: View) -> &mut Placement {
        &mut self.views[view.index()]
    }
}
```

`Index` is what `layout[view]` calls to read, and `IndexMut` what it calls to change, as in `layout[View::HarmonyKeys].shown = true`. `View` is an enum without data, so `view` as `usize` gives its position in the declaration, which `View::index` returns. Nobody can index the array with a number that is not a view.

Dragging a box changes its `rect` with `move_by`, its corner with `resize_by`, and the scroll wheel its zoom with `zoom_by`, each keeping the box inside the room and the zoom between 50% and 250%. The wheel zooms by `(scroll / 500.0).exp()`: scrolling up and back down by the same amount multiplies by a number and then by its inverse, so it comes back to exactly where it was. **Stack** calls `Layout::stack`, which puts the harmony keys at the bottom, as tall as they need, and shares the room above evenly between the other views shown.

The dialog [remembers](#) the layout and the size between runs, with egui's `get_persisted` and `insert_persisted`. `Layout` and `Placement` derive `Serialize` and `Deserialize`, and `#[serde(default)]` fills in any field missing from what was saved with its default, so settings written by another version of the app still read.

Export video

The song's sound, with the views you pick playing along.

- ☒ **Chord sheet** 250% zoom
- ☐ **Lead sheet** 100% zoom
- ☐ **Playlist** 100% zoom
- ☐ **Piano roll** 100% zoom
- ☒ **Harmony keys** 100% zoom

The starting song Bar 1, beat 1 C major · 4/4 · 130 BPM

Chord sheet · 250%

1 4/4	2	3	4
C Am	F G	C Am	F
I vi	IV V	I vi	IV
5	6	7	8
C Am	F G	C Am	F
I vi	IV V	I vi	IV
9	10	11	12

Harmony keys · 100%

Time 4/4 16 steps a bar

Triads 7ths 9ths

Short C → F: away from home

Find the key

I C tonic

ii Dm subdominant

iii Em tonic

IV F subdominant

V G dominant

vi Am tonic

vii° Bdim dominant

Stack Preview at Bar 1, beat 1

Drag a view to move it, drag its corner to resize it, and scroll over it to zoom.

Size ☒ 1280 × 720 ☐ 1920 × 1080

An MP4 file: H.264 pictures, 30 a second, and MP3 sound.

Export... Cancel

Here the chord sheet, zoomed to 140%, sits beside the lead sheet, with the harmony keys at 120% below:

El Choclo Bar 29, beat 1 A harmonic minor · 4/4 · 112 BPM

Chord sheet

E7 V7	AIII i	G7 bVII7	C bIII	G7 bVII7	C bIII	E7 V7	AIII i
13 Dm iv	E7 V7	14 G7 bVII7	C bIII	15 G7 bVII7	C bIII	16 E7 V7	Am i
17 Dm iv	E7 V7	18 A I	E7 V7	19 E7 I	A I	20 F#7 VI7	Bm ii
21 B7 II7	E7 V7	22 A I	E7 V7	23 E7 I	A I	24 F#7 VI7	Bm ii
25 B7 II7	E7 V7	A I	E7 V7	27 Am i	E7 V7	28 E7	
29 E7 i	Am i	30 Am		31 A7 I7	Dm iv	32 Dm i	Am i

Key: A harmonic minor
Time: 4/4 16 steps a bar
Triads: 7ths 9ths
Short: E7 → Am: back home

Chord functions: i (Am tonic), ii° (Bdim subdominant), bIII+ (Caug tonic), iv (Dm subdominant), V7 (E7 dominant), bVI (F tonic), vii° (G#dim dominant)

Testing it

The renderer's tests play into a `Vec` and look at the numbers:

- `renders_like_the_audio_thread`: a click rendered by the renderer is exactly the click the audio thread makes, `tanh(0.5)` on both channels, and ten frames long.
- `knows_where_playback_is`: at 120 BPM a step lasts 6,000 frames at 48 kHz, so after 9,000 frames the position is 1.5 steps.
- `exports_a_wav_file`: the file has the right format, length and first sample, `progress` was called after each block, and no `.tmp` file is left behind.
- `a_cancelled_export_leaves_no_file`: neither the file nor the temporary one exists after a stop.

The MP4 writer's tests check the bytes: `a_box_starts_with_its_size_and_name`, `samples_of_the_same_length_are_listed_as_runs`, and `writes_a_movie_the_tables_describe`, which reads back where each sample landed and how long the movie lasts. The scene's tests draw pictures and count pixels: `views_are_drawn_in_their_boxes` puts the chord sheet in the right half and checks that the left half is empty, and `zoomed_in_views_are_drawn_bigger` checks that a view at 200% has much more ink than at 100%. Two tests in the app export a whole video of the starting song with every view, and stop one halfway to check that no file is left.

What tests cannot check is whether a file plays well everywhere. For that, open an export in any audio editor: the shape of the song is easy to see in the waveform. The starting song's WAV file has exactly 739,523 frames, left and right are equal (its channels are all in the middle and the reverb is off), bars 1 and 2 hold the chords alone, bars 3 to 7 the beat, bar 8 the fill, and the tail is quiet, with no chord of the song starting again. A video should open in any player, and jumping around should start drawing at once, thanks to a key picture every two seconds.

Exercises on exporting are in [Exercises: sound](#).

The playback trace

Files: `crates/lineup-audio/src/trace.rs`, `crates/lineup-audio/src/engine.rs`,
`crates/lineup-app/src/app/trace.rs` and `crates/lineup-app/src/main.rs`

The screen shows the project; it does not show what the player did with it. A song put into another time signature can look right, every bar redrawn in 3/4, and still play its bars at the old pace, because the notes were squeezed and the tempo slowed down, and the two cancel out. Only the sound tells the truth. But listening is slow, and it says that something is wrong, not what.

A **trace** is a list of every note the player plays, written down by the audio thread at the moment it plays it. It is not worked out again from the project: it is what the speakers get, after swing, tempo changes, nudges and loose timing, with the song unrolled the way the playlist plays it and the muted tracks left out. So a trace can be read instead of listened to, by you or by a test. Many of the program's tests read traces, and so does anyone checking that a demo plays what its score says.

Trying it

From a terminal, with a song file or the name of a demo:

```
ferrous-lineup --trace song.lineup --out song.tsv
ferrous-lineup --trace "Twelve-bar blues"
ferrous-lineup --demos
```

`--trace` plays the song the way **File > Export WAV...** does, from start to end, but without a window, a sound card or a sound file: you get the notes of the export, frame for frame, in a second or a few. `--out` writes them to a file; without it they go to the terminal. `--rate` plays at another sample rate than the export's 44,100. `--demos` lists the names `--trace` understands. Capitals do not matter, and part of a name is enough when only one demo has it.

On Windows, use `--out`: the ready-built program is a windowed program and has no terminal to print to.

In the app, **Edit > Settings... > Playback > Trace** has a box, **Write a trace while playing**. While it is ticked, every playback, from Play to Stop, adds a table to `playback-trace.tsv`, in the folder the app keeps its settings in; the setting shows the full path. It starts unticked each time the app opens, so a trace you forgot about never

fills your disk. The box is greyed out when no sound card is open, since then nothing plays. The manual suggests attaching this file when reporting a problem with playback.

Reading the table

The lines starting with `#` say what the table holds, then list the tempo and the time signature of every part of the song. These are the first ones of Für Elise:

```
# Ferrous Lineup playback trace: Für Elise
# Every note the player started, in the order they start. Times are in
seconds from the start of playback;
# steps are sixteenth notes of the loop played (the song, or the pattern
in pattern mode), counted from 0.
# sample rate: 44100 Hz
# tempo: 0.0000 s, step 0, bar 1: 144.00 BPM
# tempo: 123.7500 s, step 1188, bar 100: 216.00 BPM
# tempo: 130.0000 s, step 1278, bar 105: 144.00 BPM
# tempo: 156.2500 s, step 1530, bar 126: 142.80 BPM
# tempo: 156.3551 s, step 1531, bar 126: 141.60 BPM
...
# time: step 0, bar 1: 3/4
# time: step 1188, bar 100: 9/8
# time: step 1278, bar 105: 3/4
```

The tempo changes as written at bars 100 and 105, then glides down from bar 126: a *ritardando*, which the player plays one step at a time, each a little slower, so the trace has a line for each step of it.

Then comes one line for each note, in the order they start, with a tab between columns. These are the first notes of the bass and the hi-hat in the **Twelve-bar blues** demo, with some columns left out:

start	length	bar	beat	into_beat	step	steps	bpm	name	pitch	note	velocity	late_ms	end
0.0000	0.4000	1	1	0.000	0.000	2.667	100.00	Bass	36	C2	104	0.0	gate
0.0000	0.1500	1	1	0.000	0.000	1.000	100.00	Hat	60	C4	96	0.0	gate
0.4000	0.0500	1	1	2.667	2.667	0.333	100.00	Hat	60	C4	96	100.0	gate
0.4000	0.2000	1	1	2.667	2.667	1.333	100.00	Bass	40	E2	90	100.0	gate
0.6000	0.4000	1	2	0.000	4.000	2.667	100.00	Bass	43	G2	96	0.0	gate
0.6000	0.1500	1	2	0.000	4.000	1.000	100.00	Hat	60	C4	96	0.0	gate

The swing is right there. At 100 BPM a beat lasts 0.6 seconds and has four steps. Written down, the second eighth of the beat is on step 2, halfway. Played, it comes 2.667 steps into the beat, 100 ms late: two thirds of the way, the third note of a triplet, which is what full swing means. And the bass note before it lasts 2.667 steps instead of 2, long before short. (How swing moves the notes is in [Swing](#).)

Column	What it says
<code>start</code> , <code>length</code>	When the note starts and how long it is held, in seconds from the start of playback
<code>bar</code> , <code>beat</code>	Its bar and its beat, counted from 1, in the time signature in force
<code>into_beat</code>	How many steps after the start of its beat it starts
<code>step</code> , <code>steps</code>	The step it starts on (counted from 0, with the fraction it is late by) and how many steps it is held
<code>bpm</code> , <code>time</code>	The tempo of its step and the time signature of its bar
<code>channel</code> , <code>name</code> , <code>program</code> , <code>instrument</code>	Its row in the channel rack (from 1), the channel's name, its General MIDI program and what plays it
<code>pitch</code> , <code>note</code>	The MIDI note number and its name, spelled in the key of the moment
<code>velocity</code>	How hard it is played, 1 to 127
<code>late_ms</code>	How long after its step it starts: its nudge, the swing and loose timing
<code>end</code>	Why it ended: <code>gate</code> when its length was over, <code>cut</code> when it was stopped early (Stop, a jump, its track muted, its clip taken away), <code>again</code> when the same note started again on a MIDI Out or a SoundFont, which play one of each note at a time

Column	What it says
pattern	Its pattern and the step of the pattern it comes from

Tables like this are made for `grep` and `awk`. The bass notes of bar 12, or every note louder than 110:

```
ferrous-lineup --trace song.lineup | awk -F'\t' '$11 == "Bass" && $3 == 12'
```

```
ferrous-lineup --trace song.lineup | awk -F'\t' '!/^#/ && $16+0 > 110'
```

A spreadsheet opens the file too: tell it the columns are separated by tabs, and that lines starting with `#` are comments. The table is always in English and names notes with letters, whatever the window shows, so a trace reads the same for everyone.

What a trace does not tell

A trace follows the notes, not the sound. A drum sample plays to its end whatever the note's length, a synth's release fades on after the note ends, and when every voice of the Lineup Osc is busy the oldest one is taken for the new note. None of that is in the trace. For a problem inside an instrument, listening (or looking at the samples) is still the way.

Writing on the audio thread

The audio thread has [strict rules](#): it must never wait, never allocate memory, never write to a file. Writing a trace is exactly what it may not do. So it does what it already does for MIDI Out: it hands small messages to a ring buffer, and another thread, which may take its time, reads them.

The messages are a `Copy` enum, so pushing one copies a few bytes and allocates nothing:


```

pub enum TraceEvent {
    /// Playback started on `step`: of the song, or of pattern `pattern`
    /// when `song` is false. After a count-in, when the count is over.
    Play {
        frame: u64,
        step: usize,
        song: bool,
        pattern: usize,
    },
    /// Playback jumped to `step` while playing.
    Jump { frame: u64, step: usize },
    /// Playback stopped.
    Stop { frame: u64 },
    /// A step started, at `tempo`.
    Step {
        frame: u64,
        step: usize,
        tempo: Tempo,
    },
    /// A note started on channel `channel` (its row in the channel rack),
    /// `late` frames after its step (nudge and loose timing), to be held
    /// for `gate` frames. `id` pairs it with its [`TraceEvent::NoteOff`].
    NoteOn {
        frame: u64,
        id: u32,
        channel: usize,
        origin: Origin,
        pitch: Note,
        velocity: u8,
        gate: u32,
        late: u32,
    },
    /// The note `id` ended, as `end` says.
    NoteOff { frame: u64, id: u32, end: End },
}

```

`frame` is the audio thread's own clock: the frames it has played since it started. Every note gets an `id`, so its `NoteOff` can be paired with its `NoteOn` later. The ids count up with `wrapping_add(1)`, which goes back to 0 after `u32::MAX` instead of panicking in a debug build; four billion notes later, the first ones are long finished.

The audio thread keeps the tracer as an `Option<Box<Tracer>>`. When the trace is off it is `None`, and every place that traces is an `if let Some(trace) = &mut self.trace`, which costs next to nothing. The tracer is made on another thread, where its memory can be reserved:

```

/// A tracer for the audio thread and the reader of what it writes.
pub fn trace_queue() -> (Tracer, TraceReader) {
    let (events, consumer) = rtrb::RingBuffer::new(TRACE_QUEUE_SIZE);
    let lost = Arc::new(AtomicUsize::new(0));
    let tracer = Tracer {
        events,
        lost: Arc::clone(&lost),
        open: Vec::with_capacity(OPEN_NOTES),
        next_id: 0,
        next_end: u64::MAX,
    };
    let reader = TraceReader {
        events: consumer,
        lost,
    };
    (tracer, reader)
}

```

and travels to the audio thread inside a command, the same way an instrument does: `Command::SetTrace(Some(Box::new(tracer)))`. The queue holds 65,536 events (1 << 16), and the list of notes sounding is reserved for 4,096, so nothing grows on the audio thread. An old tracer goes back to the UI thread as garbage, to be freed there.

Lost on purpose

If the ring buffer is full, because nobody read it for a long time, the event is not pushed and a counter of lost events goes up:

```

fn push(&mut self, event: TraceEvent) {
    if self.events.push(event).is_err() {
        self.lost.fetch_add(1, Ordering::Relaxed);
    }
}

```

The table's header then says how many events were lost, rather than silently missing notes. The queues that carry the *music* do the opposite: a MIDI note-off that finds its queue full waits for room, because a lost one would leave a note sounding for ever on a synth (see [When the sound card goes away](#)). A trace is a report, not the music. Making the audio thread wait for it would put the report before the sound.

Where the notes start

A note of the song starts in one of two places: in `start_pattern_notes`, when its step comes, or in `start_pending`, a few milliseconds later, for a note that is nudged or

loosely timed. Both tell the tracer on the frame the note really starts, which is why a swung note shows its 100 ms:

```
if let Some(trace) = &mut self.trace {
    trace.note_on(
        self.now,
        note.channel,
        note.origin,
        note.pitch,
        note.velocity,
        note.gate,
        note.late,
        one_per_pitch(&channel.instrument),
    );
}
```

`one_per_pitch` says whether the instrument is a MIDI Out or a SoundFont, which play one of each note at a time: on those, a note that starts again ends the one sounding, and the trace writes `again` for it.

How long a note lasts

When a note starts, its length is turned into a number of frames, the *gate*. If the tempo changes while it sounds, every instrument stretches the gates of its notes with the same function, `stretched`, so the note still ends on its step (see [Every step at its own tempo](#)). The tracer keeps its own list of the notes sounding, `open`, and stretches them with that very function:

```
pub(crate) fn tempo_changed(&mut self, now: u64, next_step: f64, stretch: f64) {
    for note in &mut self.open {
        if note.end != u64::MAX {
            let left = note.end.saturating_sub(now) as f64;
            note.end = now + stretched(left, next_step, stretch) as u64;
        }
    }
    self.find_next_end();
}
```

A note with the gate that never runs out, `HELD` (`u32::MAX`), has no end the tracer could know, which it keeps as `u64::MAX`.

The tracer also hears everything that ends notes early: Stop, a jump, a mute, a song sent while playing that no longer has a clip, a channel taken away. The song-changed case shows a small Rust detail. The audio thread checks each sounding note with a

closure, `still_played`, that borrows the song. The instruments and the tracer both need it:

```
for channel in &mut self.channels {
    channel.instrument.release_unless(still_played);
}
if let Some(trace) = &mut self.trace {
    trace.cut(self.now, None, still_played);
}
```

`still_played` is passed by value, once for each channel and then once more. That works because a closure that only captures shared references is `Copy`, like the references themselves: passing it hands over a copy and the original stays usable.

Only when a note ends

To find the notes whose length is over, the tracer would have to look at every one of them on every frame, 44,100 times a second. Instead it remembers the earliest end, `next_end`, and does nothing until that frame comes:

```
/// Ends the notes whose length is over by frame `now`. Called once
/// a frame, before anything else happens on it.
pub(crate) fn frame(&mut self, now: u64) {
    if now < self.next_end {
        return;
    }
    // The fields are borrowed one by one: `open` to go through, the
    // queue and the counter to write to.
    let events = &mut self.events;
    let lost = &self.lost;
    self.open.retain(|note| {
        if note.end > now {
            return true;
        }
        let event = TraceEvent::NoteOff {
            frame: note.end,
            id: note.id,
            end: End::Gate,
        };
        if events.push(event).is_err() {
            lost.fetch_add(1, Ordering::Relaxed);
        }
        false
    });
    self.find_next_end();
}
```

`retain` keeps the items for which the closure returns `true` and drops the others, in one pass and in place. The closure needs to push to the queue while `retain` holds `self.open` mutably. Calling `self.push` inside it would borrow all of `self` a second time, which the compiler refuses. Borrowing the two other fields into local variables first (`events`, `lost`) makes three borrows of three different fields, which do not overlap.

Turning events into notes

`Trace::notes` reads the events after the fact, on an ordinary thread. It splits them into playbacks (`runs`) at each `Play`, and keeps the `Step` events of each one in a list, in the order they were played. To find where a frame is, it looks for the last step that started before it, with `partition_point`:

```
fn position(&self, frame: u64, sample_rate: u32) -> Option<(f64, f64, Tempo)> {
    let index = self.steps.partition_point(|played| played.frame <= frame);
    let played = self.steps.get(index.checked_sub(1))?;
    let length = match self.steps.get(index) {
        Some(next) => (next.frame - played.frame) as f64,
        None => frames_per_step(played.tempo, sample_rate),
    };
    let fraction = (frame - played.frame) as f64 / length.max(1.0);
    Some((
        played.step as f64 + fraction,
        played.count as f64 + fraction,
        played.tempo,
    ))
}
```

`partition_point` is a binary search: given a sorted list and a test that is true for a first part of it and false for the rest, it returns where the false part starts. Here, that is the first step that started after `frame`; the one before it is the step being played. The fraction is how far `frame` is between that step and the next. `index.checked_sub(1)?` returns `None` for a frame before the first step, where `index - 1` would underflow.

Each step also has a `count` of the steps played before it, which goes on growing when a pattern loops or the playhead jumps. A note's length in steps is the difference of those counts: it is measured along the steps really played, whatever numbers the loop gives them. A note held across the end of a one-bar loop, from step 14 to step 2 of the next time round, lasts 4 steps, not -12.

A `NoteOff` finds its `NoteOn` by id in a small list of the notes still open in the run. `swap_remove` takes it out by moving the last item into its place, which is quick and fine here, since the order of that list does not matter.

Writing the table

`write_table` writes the header and one line per note. The names come from `Labels`: the song's title, the project (for the channels' and patterns' names), the time signatures and the keys, all in the steps of the loop that was played:

```
/// For a trace of song mode playing `arrangement`.
pub fn song(title: &'a str, project: &'a Project, arrangement:
&Arrangement) -> Self {
    Labels {
        title,
        project,
        meter: project.played_meter(arrangement),
        keys: project.played_keys(arrangement),
    }
}
```

`played_meter` and `played_keys` unroll the time signatures and keys the same way the arrangement unrolls the clips, so a section in 3/4 that plays twice is in 3/4 both times. Each note is spelled in the key of its step: an F♯ in G major, a G♭ in D♭ major.

A channel or pattern name could hold a tab or a line break, which would split the table. `cell` replaces them with spaces:

```
/// Text for a table cell: no tabs or line breaks, which would split it.
fn cell(text: &str) -> String {
    text.replace(['\t', '\n', '\r'], " ")
}
```

`str::replace` takes a *pattern*, and an array of characters is one: it matches any of them.

A note still sounding when the trace ended has no length; the table writes `?` for it, and the header says how many there are.

A trace while you play

The setting's trace is a `LiveTrace`, kept by the app. Once a frame, `check_trace` moves what the audio thread handed over into it. Each playback is written as its own table once it is over, at Stop, or when another starts (switching between song and pattern mode while playing starts another). `Trace::take_finished` takes the finished playbacks out and leaves the one still going.

A table needs the project's names, but you can rename a channel while a song plays. So at each `Play`, `check_trace` keeps a copy of the project as it was, and the table is written with the names the playback had:

```
for event in &live.trace.events[before..] {
  if let TraceEvent::Play { song, pattern, .. } = *event {
    live.playing.push_back(Playing {
      title: self
        .file
        .song_name()
        .unwrap_or_else(|| "Untitled".to_string()),
      project: self.project.clone(),
      song,
      pattern,
    });
  }
}
```

A `VecDeque` is a queue that can add at one end and take from the other: copies go in at the back as playbacks start, and come out at the front as their tables are written.

The window shows note names in letters or solfège and words in English or Spanish, but the table must not change with them. So `write` sets both for as long as it writes, with the `with` functions of the [per-thread settings](#):

```
let mut table = Vec::new();
lineup_core::i18n::Language::English.with(|| {
  lineup_core::NoteNaming::Letters.with(|| write_table(&mut table,
    trace, &labels))
})?;
```

Unticking the box, or closing the app, drops the `LiveTrace`. Its `Drop` writes the playback still going, so the last notes are not lost. If the sound card goes away and comes back, `resume_trace` writes what the old audio thread left and hands a new tracer to the new one, and the trace goes on in the same file.

The command line

`main` looks at its first argument before it opens a window (see `main.rs`). After `--trace`, a small loop reads the rest:

```
while let Some(arg) = args.next() {
    match arg.to_str() {
        Some("--out" | "-o") => out = args.next().map(PathBuf::from),
        Some("--rate") => match args.next().and_then(|hz|
hz.to_str()?.parse().ok()) {
            Some(hz) if (8_000..=192_000).contains(&hz) => rate = hz,
            _ => {
                eprintln!("--rate takes a sample rate in Hz, such as
44100");
                return 2;
            }
        },
        _ if source.is_none() => source = Some(arg),
        _ => {
            eprintln!("{USAGE}");
            return 2;
        }
    }
}
```

`args` is an iterator, and the loop takes items from it two ways: one per turn with `while let`, and one more with `args.next()` inside the body, for an option's value. The arms show three kinds of pattern: `"--out" | "-o"` matches either text, `Some(hz)` if ... adds a guard, and `_ if source.is_none()` matches anything, but only while no song was given. The function returns the exit code the program ends with: 0 when the trace was written, 1 when it could not be, 2 for a mistake in the arguments. Scripts check it.

`trace_table` builds the whole app without a window or a sound card, the way the tests do (`Audio::Unavailable`), opens the file or demo, and asks for the same song render the WAV export makes, with a trace switched on:

```
pub(super) fn trace_song(&mut self, sample_rate: u32) -> Trace {
    let mut render = self.song_render_traced(sample_rate);
    render.renderer.skip(render.frames);
    render.renderer.take_trace().unwrap_or_default()
}
```

`skip` plays all the frames and throws the sound away. The trace is the export's, because it is the same code path (see [Exporting](#)).

A demo name is matched in three tries: exactly, then ignoring capitals, then as part of a name. The last try can find none, one or several, and a slice pattern says which:

```
match matching.as_slice() {
  [demo] => Ok(*demo),
  [] => Err(format!(
    "There is no file or demo called \"{name}\" (--demos lists the
demos).",
  )),
  several => {
    let names: Vec<&str> = several.iter().map(|demo|
demo.name()).collect();
    Err(format!("\"{name}\" could be any of: {}", names.join(", ")))
  }
}
```

Tests that read instead of listen

The tests use traces the way a person would. In `lineup-audio`, on a small renderer:

- `notes_start_on_their_step_and_last_their_length`, and `a_nudged_note_starts_late_by_its_nudge`.
- `a_slower_tempo_stretches_a_held_note`: a note that sounds across a change to half the tempo lasts its steps, in more seconds.
- `stopping_and_muting_cut_the_notes_sounding`, and `a_midi_out_ends_a_note_that_starts_again`.
- `a_looping_pattern_counts_its_steps_on`: lengths are measured along the steps played, across the loop.

And in the app, on whole songs:

- `the_trace_plays_every_note_of_the_export`: every note the project holds, clip by clip, is in the trace exactly once, and all of them end when their length is over.
- `swing_plays_the_off_beats_late`: with swing, the only notes that start late are the second eighths of each beat.
- `a_tempo_marker_changes_the_seconds_not_the_steps`: a tempo marker at step 16 changes when the notes after it start, in seconds, but not their steps.
- `re_metering_keeps_each_bar_as_long_in_seconds`: a song put from 4/4 into 3/4 starts every bar at the same second as before. That is the bug from the start of this chapter, the one only the sound could show, written as a test.
- `a_trace_while_playing_writes_a_table_for_each_playback`.

Other chapters' tests use traces too, such as the one that takes El Choclo through seven styles and checks that every melody note still starts and lasts as it did ([Tempo, meter, swing and feel](#)).

Exercises on traces are in [Exercises: sound](#).

Exercises: sound

These exercises practise this part on the real code: the sequencer, tempo and meter, the instruments, the mixer, MIDI, the exports and the trace. Each one says what is **already there**, so you extend it rather than build it again. Some are things to try in the app and listen to; some are pieces of Rust with a clear end; the ones called “Think about it” are questions, with no code to write.

Try the Rust ones on a branch of your own. Run `cargo test` after each change, and `cargo fmt --all --check` and `cargo clippy --workspace --all-targets -- -D warnings` before you call one finished: CI runs all three. Several exercises grow the project file format; each needs the next free format number, so if you do two of them, the second one takes the number after the first.

Timing, tempo and meter

1. A meter map inside a pattern (lineup-core, hard)

The song already has a meter map: `Meter` keeps the starting `TimeSignature` and a sorted list of `TimeMarker`s, and `bar_at`, `time_at` and `click` read it (see [Tempo, meter, swing and feel](#)). A `Pattern` still has only one `time: Option<TimeSignature>`, so a part that changes meter every bar is cut into a pattern per run of bars.

Let one pattern hold a meter map of its own, with changes at bar lines.

`Pattern::length` counts steps, and a bar is no longer one fixed number of them, so:

- Which code that asks `TimeSignature::steps_per_bar` or `Meter::bar_at` for a pattern would have to ask the pattern’s map instead: the note tools, `Project::pattern_clicks`, the piano roll’s bar lines?
- What should the playlist and the piano roll ask when a clip starts partway through a pattern?
- How would `format_for` account for it, so a pattern in one meter is written exactly as now?

Start with `Meter` rather than inventing a second type. `bar_at` walks the stretches between markers one by one. Why can it not simply use `partition_point` on the markers, as other sorted lists in the program do? What would it need to keep beside each marker to make that possible?

Practises: reading an existing data structure before extending it, bar arithmetic, keeping files compatible.

2. Swing by ear (the app, easy)

Open *Midnight Shuffle* and play it with the **Swing** control at straight, 50 and 100. Which instrument shows the change most? Which setting sounds like a march, which like a shuffle?

Then work it out. At 96 BPM, how many milliseconds after the beat does the second eighth start at straight, 50 and 100? (A beat is 96 ticks, and `swing::delay` says how many ticks late the second eighth is.) At 120 BPM the third note of a triplet comes 333 ms after the beat: why is the 100 answer not 333 ms at 96 BPM? Check your answers with a `trace`: the `late_ms` column.

Practises: turning ticks into time.

3. Swing in Rust (lineup-core, medium)

`swing.rs` delays the second eighth of every beat as the notes are *played*: read `delay`, `swung`, `swings_in` and `Project::played_notes`.

- Write a test that a note held from the second sixteenth of a beat (step 1) to the offbeat ends 16 ticks later at full swing on a piano channel, but not on a drum channel. Which argument of `swung` makes the difference, and why should a drum hit not get longer?
- Add sixteenth-note swing: within each eighth, delay the second sixteenth (the “e” and the “a” of “1 e & a”) by a smaller amount, chosen by a second field or a `SwingUnit` enum. Which ticks move, and by how much at full swing? Write a test that pairs of sixteenths still take the same total time, and that the written notes are untouched. What must `swings_in` say about 6/8, where the sixteenths come in pairs too? What else does the new setting touch: the project file, the control, the MIDI export?

Practises: arithmetic with ticks, unit tests with exact numbers, keeping playback and editing apart.

4. Odd meters by ear (the app, easy)

- Open **Ajde Jano** and, in the Time list, type 7 over 8 grouped as 2+2+3 , and click **Use 2+2+3/8**. The melody keeps its notes, but the band now leans on other eighths. Which beats of the tune now land between the bass notes?
- In the Library, play the **Kalamatianos** and the **Rachenitsa** rhythms one after the other. Clap along: where is the long beat in each?
- Pick 5/8 in the Time list of a song in 4/4, with **Drums** and **Chord and bass parts** ticked. Which rhythm does it get, and how does the list write the meter?
- Open **Süsserl** and watch the ruler while it plays. Count the bars of two and the bars of three in the first strain: do they match 2 2 3, 2 2 3, 2 2 3 3?

Practises: hearing grouped beats, reading `TimeSignature` groups.

5. Think about it: swing

- In a song with full swing, write a dotted eighth and a sixteenth in the piano roll, then two eighths. Which pair moves? Many jazz players read both the same way, as a triplet. What would Ferrous Lineup need to know to do that, and why could it go wrong on a march?
- Put a swinging song in 6/8 with the Time box. What happens to the Swing control, and to the sound? Write the same groove in 12/8 by hand: where do the long and short notes go?
- `played_notes` returns a `Cow` . What would change if it returned a `Vec<NoteEvent>` instead? Count how many notes `diff` would copy for a song of 20 patterns and 8 channels, on every edit.

Instruments and SoundFonts

6. A low cowbell (lineup-audio and lineup-core, easy)

Read `drums::cowbell` : two square waves at 540 and 800 Hz (a little less than a fifth apart: 800 / 540 is about 680 cents, a fifth 702), a high-pass at 400 Hz, a low-pass at 3 kHz, and two decays.

Make the two frequencies parameters (`cowbell_at(sample_rate, low, high)`) and add a `Drum::LowCowbell` an octave below. The compiler lists every `match` to update in `drums::render` , `Drum::name` and `general_midi_drum` ; add the new drum to `Drum::ALL` too (and change its length), or the menus and the drum tests will not see

it. Which General MIDI note should it be sent as? (`general_midi::drum_name` lists them.)

Adding a drum also changes which projects older versions can open. Read `format_for` and `Drum::is_new_in_format_3`: is that check enough for a drum added now, or does the file need the next format number?

Practises: enums, exhaustive `match`, a change that crosses crates.

7. A legato synth (lineup-audio, harder)

The Lineup Osc plays each note on a voice of its own. Give it a mono, *legato* mode: only one voice plays, and a new note changes that voice's pitch without restarting the envelope. Add a `glide` time so the pitch slides from the old note to the new one (use `dsp::Smoothed` for the frequency). How does `pick_voice` change? Which parts of the `Stage` state machine are skipped? Where would the setting live, and how does it reach the audio thread?

Practises: state machines, reusing small DSP pieces.

8. A click on the offbeats (lineup-audio, medium)

The metronome clicks the beats, higher on the first beat of a bar: `Project::played_clicks` and `pattern_clicks` build a `Click (None, Beat or Bar)` for every step, `metronome::click_at` chooses one, and `ClickVoice` makes the sound. Add a **subdivision** setting (eighths, triplets or sixteenths) that makes it click between the beats too, quieter than the beat and with the bar accent kept. Decide whether the subdivision belongs in `Click` or in a second setting sent to the audio thread. Should the count-in stay on the beat?

The click must not be in the WAV export, or in the MIDI export. Where does each of those come from, and which one would get the click by accident if it lived in a channel?

Practises: a new setting end to end, real-time sound without allocating.

9. Another SoundFont (the app, easy)

Download another General MIDI SoundFont in `.sf2` form (*FluidR3_GM*, or the small *TimGM6mb*), pick it with **Choose SoundFont...**, and play the Canon in D. Which strings

do you prefer? Is one SoundFont much louder than the other? Where would you put a volume setting for the SoundFont as a whole: in `Sounds`, on the audio thread, or in each `SoundFontPlayer`?

Practises: listening, finding where a setting belongs.

10. An instrument for the Lead (lineup-core, easy)

Twelve-tone rows and micromodes write their melody on a channel called Lead, which `library::apply::synth_named` makes with no General MIDI program, so it keeps the Lineup Osc while the Keys get a piano and the Bass a bass. Pick an instrument for it in the rack and listen. Which instrument would you give the Lead by default, and where in `synth_named` would it go? Which tests would tell you that the songs, which choose their own Lead, still keep theirs?

Practises: reading a `match` on names, defaults.

11. Think about it: instruments

- A SoundFont renders in stereo, but `SoundFontPlayer::next_sample` folds left and right into one sample before the channel's pan. What would it take to keep its left and right apart, and what should Pan then do: move both sides, or turn one down (a *balance*)?
- rustysynth has a reverb and a chorus of its own, and `enable_reverb_and_chorus` turns them off. Turn them on and listen. Each channel would then have its own room: what does that cost, compared with the [mixer's shared reverb](#), and why do mixing desks use a send instead?
- The export runs at 44,100 frames per second even if your sound card runs at 48,000. Look at how the drums are made (`drums::render` takes the sample rate) and how the sampler plays a sound recorded at another rate. Could an export ever sound different from what you hear?

The mixer and effects

12. A constant-power pan law (lineup-core, easy)

`Channel::pan_gains` keeps the near side full and turns the far side down in a straight line. Change it to a *constant-power* law: with `angle = (pan + 1) × π / 4`, the gains

are `[angle.cos(), angle.sin()]`. Which tests fail, and why? Move a channel from L100 to R100 with each law and listen: which one stays equally loud all the way across, and which one is louder in the middle?

Practises: trigonometry on samples, tests that pin a design.

13. Meter arithmetic (lineup-app and lineup-audio, easy)

- A level of 0.1 is how many decibels? How far is the meter lit for it? Work it out, then check with `lit` and `FLOOR_DB` in `meters.rs`.
- `levels::raise` compares `peak > 0.0` rather than `peak >= 0.0`. What would `fetch_max` do with the bits of `-0.0`, or of a NaN? (Look at how `f32::to_bits` orders them as integers.)

Practises: decibels, floats as bits.

14. Delays and compressors by ear (the app, easy)

- Put a delay of a dotted eighth with 30% mix on the Lead of a demo, and play it at 60 BPM and at 140 BPM. How many milliseconds apart are the echoes each time? Check with `Delay::seconds`.
- Set a compressor at `-30 dB` and `10:1` on the drums. Then raise its **Gain** until the drums are as loud as before. What happened to the difference between the loud and the soft hits? Why do drums with a strong compressor sound “squashed”?

Practises: listening for what an effect does.

15. A wider or narrower bell (lineup-core, lineup-audio and lineup-app, medium)

The equalizer’s middle band is a bell with a fixed width, a Q of 1 (`Biquad::set` in `lineup-audio/src/effects.rs`). Add a `width` to `Equalizer` and a slider for it, and use it in `Biquad::set`. What does the Cookbook say alpha is for a given Q? A song that sets the width needs the next free format number: teach `format_for`, and leave the field out of the file when it has its default.

Then write a test in the style of the existing ones: a sine at the bell’s centre is boosted by the gain, and one two octaves away is boosted less with a narrow bell than with a wide one.

Practises: a filter formula, a new saved field end to end.

16. A shared delay send (lineup-core, lineup-audio and lineup-app, hard)

Each channel already has a delay of its own among its effects. Add a second **send** beside the reverb's, into one delay that every channel shares, so several channels echo together. This crosses the project file, the audio thread and the mixer window.

1. In `Channel`, add `delay_send: f32`, default `0.0`, and include it in the range checks and in `has_plain_mix`. Give every constructor the default.
2. In `Mixer`, add the delay's own settings: time in steps (so it follows the tempo, as `Delay` does), feedback, and mix. Validate them in `Mixer::valid`.
3. Add `Command::SetSharedDelay` and `Command::SetDelaySend`, and a shared delay that allocates its buffer before it reaches the audio thread, never in `process`. The channel delay's `DelayLine` shows how. Follow the reverb's pattern for the send.
4. Add a **Delay** slider to each mixer line, next to **Reverb**, and a small delay section beside the reverb's controls.
5. Teach `format_for` that a non-zero delay send or non-default delay settings need the next free format number, and leave the fields out of the text when they are at their defaults.
6. Test the audio path (a send of 0 makes no echo, a send above 0 makes one) and the file (an old file opens with send 0; a song with a delay send is written in the new format and round-trips).

Done when the slider is audible in the app and in a WAV export, the audio thread allocates nothing while it runs, and a song without the delay is still written in the format it was.

Practises: everything in this part at once.

17. Score order (the app and lineup-core, easy)

- Open *In the Mood*, then *Sing, Sing, Sing*, with the mixer open. In which section is each song's "Horns" channel? Look at the Instrument column to see why.
- Open *Danny Boy*, right-click its Lead in the channel rack and pick **Instrument > Pipe > 74 Flute**. Where does it move in the mixer?
- In `score_order.rs`, the saxophones are between the clarinets and the bassoons, as most orchestral scores have them. In a big band score the

saxophones come first, above the trumpets. What would you change to get the big band order, and which test would tell you that you broke the orchestral one?

Practises: data that decides a layout.

18. Think about it: dynamics

- The compressor's `Follower` rises in 10 ms (`ATTACK_SECONDS`) instead of following every wave. Write a test that sends a 50 Hz sine through a compressor whose follower is much faster, and measure how far the output's shape is from a sine. What does that tell you about the 10 ms?
- Many desks let a compressor listen to one channel while it turns another down: a bass that ducks each time the kick drum hits ("side-chain compression"). Sketch what the channel effects would need for that. Which channel's sample would the follower read, and in which order would the audio thread have to mix the channels?

MIDI

19. Octave keys (lineup-app, easy)

The `typing keyboard` covers C4 to E6. Add two keys that move it down and up by an octave, such as `-` and `=` (`Key::Minus` and `Key::Equals`), which play no note yet. Keep the shift between C1 and C7, and show it in the MIDI menu, next to **Typing keyboard to piano**.

The tricky case: hold Z, press `=`, then let go of Z. Which note must end? Look at `typing_down`, which keeps the note each key played, not only the key. Why was it written that way, when the note could be worked out again from the key?

Practises: `Note` arithmetic, remembering what was done instead of working it out again.

20. Softer and louder typing (lineup-app, easy)

Typed keys play at `DEFAULT_VELOCITY`, so takes recorded from the typing keyboard all have the same velocity, while a MIDI keyboard's keep how hard each key was struck.

Add a “Typing velocity” slider to the MIDI menu (`egui::Slider::new(&mut value, 1..=127)`), and make Shift play at 127 while it is held.

Shift does not change `physical_key`, so the piano keeps working. But look at the filter in `typing_keyboard`: does a key that went down with Shift come up properly when Shift is let go first?

Practises: egui sliders, keyboard modifiers.

21. Notes across the loop (lineup-core, medium)

`recorded_note` has two limits. A note held across the end of the loop is cut there, and a key held for more than a whole loop is recorded as a short note.

- Make a key held across the loop end record **two** notes: one to the end of the pattern, and one from step 0 for the rest. The function then returns a `Vec<NoteEvent>` (or an array and a count). Write the tests first: (14, 2) in a 16-step pattern gives (14, 2) and (0, 2).
- For long notes, the position alone is not enough. Have the audio thread also count how many times the loop has gone round, or publish the number of steps played since Play instead of the position in the loop. Which is simpler to share through one `AtomicU64`? Which one can `finish_recording` still use?

Practises: changing a function’s return type and its callers, tests as a specification.

22. Record in song mode (all three crates, harder)

Rec switches to pattern mode, because a take goes into the pattern that loops. In song mode, the position is a step of the *song*, so a key could go into the clip playing at that moment, at `step - clip.start`.

Work out the questions before writing code. Two clips can play at once, on different tracks: which one gets the note? What if no clip plays there: should a new clip be made? A clip may be played several times in the song, and recording into one of them changes all of them, because they are the same pattern. Is that what you would want? How do other studios you know handle it?

Practises: designing behaviour before code.

23. The other pieces of a format 2 file (lineup-core, harder)

`midi_file::read::read` turns a file's bytes into events, and

`midi_file::import::import` makes a project of them. Two corners are still missing.

A **format 2** file is not one song but several independent pieces, one per track. `import` keeps only the first track. Add a way to choose a piece: list them in the notice, ask in a dialog, or import each as its own project. What should the tests say about a format 2 file with three pieces?

The other is the **SMPTE offset** meta event (`FF 54`): five bytes that say where a track starts on a timecode, for music written against film. The header's SMPTE timing (`Timing::PerSecond`) already works; this event is read past and dropped by `meta_event` . Read it and decide what a project with no video should do with it. Test both cases with small files built in the test module, the way the existing tests do.

Practises: parsing binary data with slice patterns, designing a choice.

24. Latency compensation (lineup-audio and lineup-app, harder)

[Latency](#) makes every recorded key a little late. `cpal` knows by how much: the audio callback gets a `&cpal::OutputCallbackInfo` , whose `timestamp()` has two instants, `callback` (now) and `playback` (when this block will be heard). The difference between them is the output latency.

The callback ignores that argument (`_: &cpal::OutputCallbackInfo` in `build_stream`). Pass the latency to the audio thread, publish it as a number of steps in another atomic, and subtract it from each key's position before `recorded_note` rounds it. A position just after step 0 minus the latency must go round to the end of the loop, not below zero.

Then think about the other direction. MIDI Out messages leave when their block is made, but the built-in instruments' sound of that same block is heard a little later. So an external synth plays slightly *before* the rest. How could the sending thread hold each message back by the latency? What would each message need to carry?

Practises: reading a crate's documentation (`cargo doc --open -p cpal`), time as data.

25. A MIDI note-off test (lineup-audio, medium)

Prove that **Stop** sends note-offs for notes the sequencer started on a MIDI Out channel. Read `midi_out_channels_send_their_notes` and

`stopping_releases_held_notes` in the engine's test module, which has the helpers `add_midi_out`, `received` and `note_off`. Add `stopping_releases_midi_out_notes`: a long note on MIDI channel 3, **Play**, render one step and check the NoteOn; send **Stop**, render a few frames, and check the NoteOff. Make the note long enough that it cannot have ended by its own gate.

Run the test once with its `Command::Stop` line removed, to see it fail, then put the line back. A test you have never seen fail may not be testing anything.

Practises: the engine's test harness, regression tests.

26. MIDI Out from the menus (the app, easy)

- Send the Bass of a band demo over MIDI on channel 2, then apply another style from the Library. How many Bass channels are there afterwards? Read `find_synth` in `library/apply.rs` to see why.
- Which General MIDI drum is note 60? A drum written on C4 and sent as written, rather than as its own note, would play it. Find it with `general_midi::drum_name`.
- `Channel::send_over_midi` clamps the channel with `.min(15)` instead of returning an error. Where else does a number from a menu or a file get clamped like this, and where is it refused instead? Which is better for a number read from a file?

Practises: reading code to explain behaviour.

27. Think about it: MIDI

- The live queue holds 256 messages, and `LiveInput::send` returns `false` when it is full. `on_message` ignores that. Which is worse to lose, a key-down or a key-up? Could a design make a lost key-up impossible, for example by sharing *which keys are down* (128 bits) instead of sending events?
- The sending thread wakes up a thousand times a second, even when no channel is sent over MIDI. Could it sleep longer when there is nothing to send? What would a longer sleep cost when there is?
- `release_live_keys` lets go of every key when a channel is inserted or removed. The other way would be to renumber `live_keys` to follow the channels. Which is easier to get right? Would a player ever notice the difference?
- `midir` gives `on_message` a timestamp in microseconds, which the code ignores in favour of the audio thread's position. When would `midir`'s timestamp be more precise? Why can the two clocks not simply be compared?

- A pattern played by a MIDI Out channel and exported as a `.mid` file sends the same notes in both cases. Could the export and the audio thread share more code? What would that code need to know about time?

Exporting

28. Export one pattern as audio (lineup-app, easy)

File > Export pattern as MIDI... writes the selected pattern as a MIDI file. Add the same as *audio*: an **Export pattern as WAV...** item, saving the selected pattern, played four times, as a WAV file.

The obvious plan is pattern mode: `SetMode(PlayMode::Pattern)`, `SelectPattern`, and render four pattern lengths plus the tail. But pattern mode loops the pattern for ever, so the tail would hold the pattern starting a fifth time: the problem [the export](#) solves with silent steps, and pattern mode has no song length to stretch. Instead, make a *copy* of the project, replace its playlist with four clips of the pattern on one track, and export that copy with the code you already have. What has to change in `song_render_with` so it can render a project other than `self.project`?

Practises: working on a copy, making a function more general.

29. Export the loop region (lineup-app, medium)

Songs can have a loop region, and song mode plays only that part. Add **File > Export loop region...**, which writes just the region to a WAV file.

`Project::playback_arrangement()` already returns exactly the part song mode plays, with repeats unrolled. Make `song_render_with` take an `Arrangement` instead of always calling `project.arrangement()`, and work out the frame count from that arrangement's own length and tempos. What should the file name be? What should the menu item do when there is no loop region?

Practises: reusing a calculation, thinking about an empty input.

30. Stems: one file per channel (lineup-app, medium)

Musicians often want *stems*: the song exported once per channel, each file holding only that channel, to mix elsewhere. Add **File > Export stems...**, which asks for a

folder and writes `Kick.wav`, `Clap.wav` and so on.

Each stem is an export with every other channel muted (`Command::SetMuted`). Rendering them one after the other is simplest; rendering them on several threads at once is faster on a computer with several cores. The video export's canvas already paints bands of rows with `std::thread::scope` : could the stems do the same, or do they need `thread::spawn`, given that the window must not wait for them? How should one progress bar show several exports? And what about two channels with the same name, or a name with a `/` ?

Practises: threads, file names from user text.

31. Two views in a video, and a sync check (the app, medium)

1. Open **El Choclo** and choose **File > Export WAV...**; save `el-choclo.wav`.
2. Choose **File > Export video...**, tick **Chord sheet** and **Piano roll**, put them side by side on the preview, pick **1280 × 720**, and export `el-choclo.mp4`.
3. If `ffprobe` is installed, compare the durations:

```
ffprobe -v error -show_entries format=duration -of default=nw=1:nk=1
el-choclo.wav
ffprobe -v error -show_entries format=duration -of default=nw=1:nk=1
el-choclo.mp4
```

4. Watch a bar where the chord changes. The framed bar on the chord sheet, the playhead in the piano roll and the sound should meet there together. Check a second point, such as the first bar after a repeat.

Done when the durations differ by no more than a picture or two, the two views show the same bar at the same moment, and you can say why `start_video` draws each picture **before** it renders the sound that plays under it.

Practises: checking a design with outside tools.

32. Think about it: exporting

- The progress counter uses `Ordering::Relaxed`. Suppose the window opened the file the moment the counter reached the total, without waiting for `join`. What could it find? (Look at what `export_wav_cancellable` does after its last

`progress` call.) Would a stronger `ordering` fix that, or is the problem somewhere else?

- A video's sound is MP3 at 256 kbit/s, while the WAV file is the exact samples. Could the video carry the samples unchanged instead? Which MP4 box would change, and which players would no longer play it?
- `repeat_picture` makes the last picture last longer when nothing changed. What would go wrong if the scene compared *pixels* instead of triangles to decide that? What would it cost?

The playback trace

33. Trace a bar inserted and deleted (the app and a terminal, easy)

The chord sheet and the playlist have **Insert a bar before**, **Insert a bar after** and **Delete this bar**; `--trace` writes every note the player plays.

1. Write a trace of a demo:

```
cargo run --release -- --trace "El Choclo" --out before.tsv
```

2. Open the same demo, save it as `inserted.lineup`, right-click bar 3 on the chord sheet and choose **Insert a bar after**. Save, and trace the saved file:

```
cargo run --release -- --trace inserted.lineup --out after.tsv
```

3. Compare the two tables, ignoring the `#` lines:

```
diff <(grep -v '^#' before.tsv) <(grep -v '^#' after.tsv) | head -40
```

Every note that started at or after bar 4 should now be one bar later; notes in bars 1 to 3 should be unchanged.

4. Delete the bar you inserted (it is bar 4), save, trace again, and compare with `before.tsv`.

Done when the inserted trace shows the one-bar shift in `start`, `bar` and `step`, the deleted trace matches the first one, and you can point to the code in `bars.rs` that moves the clips, markers and sections.

Practises: checking an edit by what it plays.

34. Reading traces (a terminal, easy)

- Trace a demo with a ritardando (a tempo marker that glides) and read the `# tempo` lines. How many are there, and why one per step?
- Turn **Loose** up in a song, trace it, and use `awk` to find the latest note (the `late_ms` column). Is it ever later than the setting?
- With **Write a trace while playing** on in the app, play a pattern, mute its channel in the middle of a long note, and stop. Find the note in the file: what does its `end` column say?

Practises: `awk` on tables, reading a report.

35. Think about it: the tracer

- The tracer looks for the notes whose length is over at the very top of each frame, before the tempo can change on that frame. What would go wrong with a note that ends on the same frame as a tempo change, if it looked afterwards?
- A trace loses events when its queue is full, and says how many. Could it instead keep a few events that matter more (the `NoteOff`s, say) and drop others first? What would the reader then have to cope with?

Pitch classes, notes and spelling

Files: `crates/lineup-core/src/pitch.rs` and `crates/lineup-core/src/harmony/pitch_set.rs`

This part of the book is about the music theory inside the program: scales, chords, numerals, sets, rows and the rules of voice leading, each one a few small Rust types with methods, tested like any other code. The theory itself is taught in the user manual's [Learning harmony](#) chapters; each chapter here says the musical idea in a few lines and then shows how the code holds it.

Everything else is built on the types of this chapter: a note, the twelve keys of an octave, the name a key is written with, and a set of keys.

The music, briefly

A piano has 88 keys but only twelve different notes, which repeat in every octave. The distance from one key to the next, white or black, is a **semitone**:

C	C#	D	D#	E	F	F#	G	G#	A	A#	B	C
0	1	2	3	4	5	6	7	8	9	10	11	12

A note's name without its octave is its **pitch class**: every C on the piano has pitch class C. Harmony mostly works with pitch classes; a C major chord is C, E and G, in any octave and any order. The octave is written after the name: **C4** is middle C and **A4** the A an orchestra tunes to, 440 Hz.

Each black key has two names: the key between C and D is C# and also D $\flat$. Which name is right depends on the music around it, and the rule is that a scale uses each of the seven letters once. F major has a B $\flat$, never an A#, because it already has an A. F# major's seventh note is E#, the key usually called F, because F# already has the letter F. Now and then a note needs two signs: the seventh of G# harmonic minor is F $\sharp\sharp$ (F double sharp, the key usually called G), and the seventh of a C diminished seventh chord is B $\flat\flat$ (B double flat, which sounds as A). The manual's [Notes, intervals and scales](#) tells this more slowly.

So the code needs two different things: which **key** of the piano a note is, and what it is **called**. They are different types.

Twelve pitch classes: an enum

```
#[derive(Debug, Clone, Copy, PartialEq, Eq, PartialOrd, Ord, Hash,
Serialize, Deserialize)]
pub enum PitchClass {
    C,
    CSharp,
    D,
    DSharp,
    E,
    F,
    FSharp,
    G,
    GSharp,
    A,
    ASharp,
    B,
}
```

There are exactly twelve pitch classes, so an enum fits: a `PitchClass` can never be 13 or -1. The variants carry no data (a **fieldless** enum), and Rust numbers them from 0 in the order they are written. The `as` keyword turns such an enum into its number, which is the number of semitones above C:

```
pub fn semitone(self) -> u8 {
    self as u8
}
```

Going the other way needs care, because not every number is a pitch class.

`from_semitone` wraps first, then looks the answer up in the `ALL` array, so it cannot fail:

```
pub fn from_semitone(semitone: u8) -> Self {
    Self::ALL[(semitone % 12) as usize]
}
```

`transpose` adds semitones and wraps around the octave. It takes an `i32` so it can also go down. For a negative number `%` gives a negative remainder (`-1 % 12` is `-1` in Rust), so it uses `rem_euclid`, which always gives 0 to 11:

```
pub fn transpose(self, semitones: i32) -> Self {
    let wrapped = (self.semitone() as i32 + semitones).rem_euclid(12);
    Self::from_semitone(wrapped as u8)
}
```

`semitones_up_to` answers the opposite question, how far up one pitch class is from another. From A up to C is 3 semitones (A#, B, C), and from C up to A is 9; adding 12 before subtracting keeps the `u8` from going below zero:

```
pub fn semitones_up_to(self, other: PitchClass) -> u8 {
    (other.semitone() + 12 - self.semitone()) % 12
}
```

Matching on two values at once

The everyday name of a pitch class depends on two things: which one it is, and whether the black keys are written as sharps or as flats. `match` can look at both at once when they are put in a **tuple**:

```
pub fn name(self, spelling: Spelling) -> &'static str {
    use PitchClass::*;
    use Spelling::*;
    match (self, spelling) {
        (C, _) => "C",
        (CSharp, Sharps) => "C#",
        (CSharp, Flats) => "Db",
        (D, _) => "D",
        // ...
        (B, _) => "B",
    }
}
```

- `use PitchClass::*;` inside the function lets it write `C` instead of `PitchClass::C`, without spreading those short names over the whole file.
- `_` matches anything: a white key has the same name either way.
- The compiler checks that the arms cover all 24 combinations. Leave out `(GSharp, Flats)` and it refuses to build, naming the missing case.
- `&'static str` is a borrowed string that lives as long as the program: string literals are stored in the program itself.

`Spelling` is a two-variant enum, `Sharps` (the `#[default]`) and `Flats`. Which one a key uses, and the finer choice of a letter for each note, is in [Scales, modes and keys](#).

`is_black_key` uses the `matches!` macro, a short way of asking “does this value match this pattern?” as a `bool`. The `|` means “or”:

```
pub fn is_black_key(self) -> bool {
    use PitchClass::*;
    matches!(self, CSharp | DSharp | FSharp | GSharp | ASharp)
}
```

`PitchClass` also has a `Display` implementation, so `{}` in `format!` writes it with sharps, in letters or in solfège as the settings say ([Two languages: translations, and letters or solfège](#)).

A note: a MIDI number in a type of its own

A note with its octave is stored as its MIDI note number, 0 to 127, wrapped in a struct of its own:

```
#[derive(Debug, Clone, Copy, PartialEq, Eq, PartialOrd, Ord, Hash,
Serialize, Deserialize)]
#[serde(from = "u8", into = "u8")]
pub struct Note(u8);
```

A struct with one field used to give a value a type of its own is a **newtype**. A `Note` is a `u8` in memory, but the compiler will not let a note be passed where a velocity (also a `u8`) is expected, or the other way round. The field is private, so the only ways to make a note are the ones `Note` offers, and they keep it in range:

```
pub fn new(midi: u8) -> Self {
    Note(midi.min(Self::MAX.0))
}
```

The same goes for files. `#[serde(from = "u8", into = "u8")]` tells `serde` to save a note as its number and to read it back through `From<u8>`, which calls `Note::new`, so not even a hand-edited project file can hold a note above 127:

```
impl From<u8> for Note {
    fn from(midi: u8) -> Self {
        Note::new(midi)
    }
}
```

Some notes have names in the code: `Note::C4` is middle C (60), `Note::A4` the tuning A (69), `Note::MIN` and `Note::MAX` the ends of the range.

Octaves

The pitch class is the number modulo 12, and the octave is the number divided by 12, minus one, because MIDI note 60 is middle C and middle C is called C4:

```
pub fn octave(self) -> i8 {
    (self.0 / 12) as i8 - 1
}
```

So the lowest MIDI note is C-1 and the highest G9. The octave goes with the letter, not with the key: it changes between B and C, so the key a semitone above B3 is C4.

Moving a note: Option for “it might not exist”

A pitch class always has a pitch class seven semitones above it; a note does not. Seven semitones above G9 is past the top of MIDI. `transpose` returns an `Option<Note>`, and `None` when the new note would be out of range:

```
pub fn transpose(self, semitones: i32) -> Option<Note> {
    let midi = self.0 as i32 + semitones;
    (0..=Self::MAX.0 as i32)
        .contains(&midi)
        .then_some(Note(midi as u8))
}
```

`contains` asks whether a range holds a number, and `then_some` turns a `bool` into an `Option`: `Some` of the value if true, `None` if false. The callers decide what an impossible note means: a chord built near the top of the keyboard leaves its highest notes out, and the transpose tool refuses to move a selection at all if one of its notes would fall off the keyboard, so no note is lost ([Editing tools as plain functions](#)).

The frequency

Ferrous Lineup tunes the way pianos are tuned, in **twelve-tone equal temperament**: every semitone multiplies the frequency by the same number, the twelfth root of two (about 1.0595), so twelve semitones exactly double it. A4 is 440 Hz:

```
pub fn frequency(self) -> f32 {
    let semitones_from_a4 = self.0 as f32 - Self::A4.0 as f32;
    440.0 * 2.0_f32.powf(semitones_from_a4 / 12.0)
}
```

Middle C is nine semitones below A4, so its frequency is $440 \cdot 2^{(-9/12)}$, about 261.63 Hz. The synth plays this frequency, and a sampler plays a recording faster or slower by the same rule ([Instruments: drums, sampler and Lineup Osc](#)).

Letters and names

What a note is called is a letter and some sharps or flats. The letters are an enum of their own, the seven white keys:

```
[derive(Debug, Clone, Copy, PartialEq, Eq, PartialOrd, Ord, Hash)]
pub enum Letter {
    C,
    D,
    E,
    F,
    G,
    A,
    B,
}
```

Letters go round in sevens as pitch classes go round in twelves, so they have the same kind of methods: the white key's semitone above C, read from a table, and moving up or down with `rem_euclid`:

```
pub fn semitone(self) -> u8 {
    [0, 2, 4, 5, 7, 9, 11][self as usize]
}

pub fn up(self, steps: i32) -> Letter {
    Self::ALL[(self as i32 + steps).rem_euclid(7) as usize]
}
```

Two letters above A is C (A, B, C), and `steps_up_to` counts the other way: from A to C is 2.

A name is a letter and a count of signs, positive for sharps and negative for flats:

```
[derive(Debug, Clone, Copy, PartialEq, Eq, Hash)]
pub struct NoteName {
    pub letter: Letter,
    /// Sharps if above zero, flats if below: 1 is "#", -2 is "bb".
    pub accidental: i8,
}
```

One `i8` holds both directions, so a name can never be “a sharp and a flat” at once, and `F♯` is just `NoteName::new(Letter::F, 2)`. The key of the piano a name stands for is its letter’s white key moved by the signs:

```
pub fn pitch_class(self) -> PitchClass {  
  
    PitchClass::from_semitone(self.letter.semitone()).transpose(i32::from(self  
        .accidental))  
}
```

A name for a key, with a given letter

Most of the time the code knows the key and the letter it wants, and needs the signs. B with the letter C is C_b (one flat); F with the letter E is $E^\sharp$ (one sharp). `with_letter` counts how far up from the letter’s white key the pitch class is, and takes the shorter way round: up to five semitones up is that many sharps, six or more is flats below:

```
pub fn with_letter(pitch_class: PitchClass, letter: Letter) -> Self {  
    let up = (pitch_class.semitone() + 12 - letter.semitone()) % 12;  
    // Up to five semitones up is sharps; six or more is flats below.  
    let accidental = if up <= 5 { up as i8 } else { up as i8 - 12 };  
    NoteName::new(letter, accidental)  
}
```

G with the letter F is two up, $F^\sharp$; A with the letter B is ten up, so two below, B_b .

Two simpler names are built on it. `plain` takes the letter from the everyday name, so it gives $C^\sharp$ or D_b for the black key and the white keys’ own letters. `chromatic` is the name used when there is no key to follow, as in twelve-tone music: C, $C^\sharp$, D, E_b , E, F, $F^\sharp$, G, A_b , A, B_b , B, with sharps for two black keys and flats for the other three:

```
pub fn chromatic(pitch_class: PitchClass) -> Self {  
    let spelling = match pitch_class {  
        PitchClass::CSharp | PitchClass::FSharp => Spelling::Sharps,  
        _ => Spelling::Flats,  
    };  
    NoteName::plain(pitch_class, spelling)  
}
```

Choosing the letter is the key’s job: each note of a key takes the next letter, and a note from outside the key takes the letter musicians most often use for it. That is

`Scale::spell`, in [Scales, modes and keys](#).

Intervals count letters too

An interval has a size in semitones and a size in letters. A major third is four semitones and two letters: above C it is E, above A it is C $\sharp$ (not D $\flat$, which is three letters up), and above E $\flat$ it is G. `above` moves the key by the semitones, the letter by the letters, and lets `with_letter` find the signs:

```
pub fn above(self, semitones: u8, letters: u8) -> NoteName {
    let pitch_class = self.pitch_class().transpose(i32::from(semitones));
    NoteName::with_letter(pitch_class, self.letter.up(i32::from(letters)))
}
```

The diminished seventh is nine semitones and six letters: above B it is A $\flat$, and above C it is B $\flat\flat$. That is how every chord is spelled from its root ([Chords: building, naming and inversions](#)), and how a chart is moved to another key, letter by letter, the way a musician transposes it.

A transposed chord symbol sometimes lands on a name with two signs. Moving a chart from E minor to G $\sharp$ minor moves everything up four semitones and two letters, so the D $\sharp^{\circ}7$ of E minor becomes F $\times^{\circ}7$. That is right on a score, but a chord symbol is read at a glance, so symbols take the simpler name of the same key:

```
pub fn simplified(self) -> NoteName {
    if self.accidental.abs() < 2 {
        return self;
    }
    let spelling = if self.accidental > 0 {
        Spelling::Sharps
    } else {
        Spelling::Flats
    };
    NoteName::plain(self.pitch_class(), spelling)
}
```

E $\flat\flat$ becomes D, F $\times$ becomes G, and the D $\sharp^{\circ}7$ above becomes G $^{\circ}7$.

The octave of a spelled note

A note's own name, `Note::name`, uses the everyday name, so middle C is always "C4". But when a key spells middle C as B $\sharp$, it must be written "B $\sharp$ 3": the octave number goes with the letter, and B $\sharp$ 3 is the B of the octave below, raised a semitone. In the same way, the B below middle C written as C $\flat$ is "C $\flat$ 4". `name_as` takes away the letter and the signs to find the C at the bottom of the letter's octave, and numbers the octave from there:

```
pub fn name_as(self, name: NoteName) -> String {
    let natural =
        i32::from(self.0) - i32::from(name.letter.semitone()) -
        i32::from(name.accidental);
    format!("{name}{}", natural.div_euclid(12) - 1)
}
```

For middle C written B $\sharp$, that is $60 - 11 - 1 = 48$, the MIDI number of C3, so the name is B $\sharp$ 3. The voice-leading check and the ear trainer write notes this way.

Writing and reading a name

Writing a name repeats the sign as many times as the count says:

```
pub fn format(self, naming: NoteNaming) -> String {
    let sign = if self.accidental > 0 { "#" } else { "b" };
    let accidentals = sign.repeat(self.accidental.unsigned_abs() as
    usize);
    match naming {
        NoteNaming::Letters => format!("{sign}{accidentals}",
    self.letter.as_char()),
        NoteNaming::Solfege => format!("{sign}{accidentals}",
    self.letter.solfege()),
    }
}
```

`unsigned_abs` turns -2 into 2 as an unsigned number, which can never overflow (`abs` on an `i8` would for -128). With no signs, `repeat(0)` gives an empty string.

Reading is the other way, and what the app writes it must be able to read back. In G $\sharp$ harmonic minor the chord on the seventh note is F $\times$ °7, saved as `F##dim7`, so `read` takes up to two signs, both in the same direction. It reads a name at the start of a text and returns the rest, so `"F#m7"` gives F $\sharp$ and `"m7"`:

```

let sign = |c| match c {
    '#' | '♯' => 1,
    'b' | '♭' => -1,
    _ => 0,
};
let mut accidental = 0;
for _ in 0..2 {
    let mut chars = rest.chars();
    let step = chars.next().map_or(0, sign);
    if step == 0 || (accidental != 0 && step != accidental) {
        break;
    }
    accidental += step;
    rest = chars.as_str();
}

```

- `sign` is a closure kept in a variable. Its argument's type is not written: the compiler works it out from how the closure is used, on characters.
- `chars()` walks the text one character at a time, and `as_str()` gives back what has not been walked yet. That is the safe way to step over “♯”, which takes three bytes in UTF-8, where cutting the string at a byte count could cut a character in half.
- `map_or(0, sign)` gives 0 when the text has ended, and the sign of the next character otherwise.
- The loop stops at a character that is not a sign, at a sign in the other direction, and after two signs. So “Ebbb” reads E♭ and leaves “b”, and “F♯b” reads F♯ and leaves “b”.

The letter itself must be a capital. `Letter::from_char` refuses ‘c’, because in a chord symbol a small “b” is a flat: “Bb” is B♭, not two B’s. The same reading in solfège, “Sib” for B♭, and how the app reads a name typed either way, are in [Two languages: translations, and letters or solfège](#).

A set of pitch classes in twelve bits

Chords and scales are sets of pitch classes. A set of at most twelve things fits in the bits of one integer: bit 0 means “C is in the set”, bit 1 “C♯ is in the set”, and so on up to bit 11 for B.

```

pub struct PitchSet(u16);

```

```
pub fn insert(&mut self, pitch_class: PitchClass) {
    self.0 |= 1 << pitch_class.semitone();
}

pub fn contains(self, pitch_class: PitchClass) -> bool {
    self.0 & (1 << pitch_class.semitone()) != 0
}
```

`1 << n` is the number with only bit `n` set (1 shifted left by `n` places). `|=` sets that bit, and `&` keeps only the bits two numbers share, so `x & (1 << n) != 0` asks whether bit `n` is set. A C major chord, C, E and G, is bits 0, 4 and 7:

bit:	11	10	9	8	7	6	5	4	3	2	1	0	
	B	A#	A	G#	G	F#	F	E	D#	D	C#	C	
C major:	0	0	0	0	1	0	0	1	0	0	0	1	= 0b0000_1001_0001

Other set questions are one instruction each. `count_ones()` counts the members, and `a & !b == 0` asks whether every member of `a` is also in `b` (`!` flips every bit, so `a & !b` is what `a` has and `b` lacks):

```
pub fn is_subset_of(self, other: PitchSet) -> bool {
    self.0 & !other.0 == 0
}
```

That is how the code asks whether a chord belongs to a key: A minor's A, C and E are all in the C major scale, E major's G# is not. And because a set is one small number, comparing two chords is a single `==`, and a set can be a key in a `HashMap`.

`iter` gives the members from C upwards by keeping the pitch classes the set contains:

```
pub fn iter(self) -> impl Iterator<Item = PitchClass> {
    PitchClass::ALL
        .into_iter()
        .filter(move |&pitch_class| self.contains(pitch_class))
}
```

`move` gives the closure its own copy of `self` (a `PitchSet` is `Copy`), because the iterator is returned and outlives the call.

Seeing a chord from its root

The trick behind naming chords is moving a whole set up or down. In bits, that is a **rotation** of the twelve bits: the ones that go past B come back in at C.

```
pub fn transpose(self, semitones: u8) -> PitchSet {
    let shift = (semitones % 12) as u32;
    let bits = self.0 as u32;
    // Rotate the 12 bits left by `shift`: bits that go past B come
    // back in at C.
    let rotated = (bits << shift) | (bits >> (12 - shift));
    PitchSet(rotated as u16 & ALL_BITS)
}
```

`bits << shift` moves every member up; the ones that went past bit 11 are dropped by the mask `ALL_BITS` (twelve ones). `bits >> (12 - shift)` brings exactly those back in at the bottom. C major moved up by 9 has A (bit 9), C# (bit 13, back in as bit 1) and E (bit 16, back in as bit 4): A major.

`relative_to` moves a set down until a given root becomes C:

```
pub fn relative_to(self, root: PitchClass) -> PitchSet {
    self.transpose(12 - root.semitone())
}
```

E minor, E G B, seen from E is C D# G: semitones 0, 3 and 7, the shape of every minor triad. Seen from its root, every chord of one kind is the same number, whatever its root. [Chords: building, naming and inversions](#) names a chord by finding the root from which its set matches a known shape.

Making collect build a PitchSet

`collect()` can build any type that implements the `FromIterator` trait; `Vec`, `String` and `HashMap` do. Implementing it for `PitchSet` lets any code turn notes into a set in one line, such as `notes.map(|note| note.pitch_class()).collect::<PitchSet>()`:

```
impl FromIterator<PitchClass> for PitchSet {
    fn from_iter<I: IntoIterator<Item = PitchClass>>(iter: I) -> Self {
        let mut set = PitchSet::EMPTY;
        for pitch_class in iter {
            set.insert(pitch_class);
        }
        set
    }
}
```

`from_semitones(&[0, 4, 7])` builds a set from numbers the same way, by mapping each one to a pitch class and collecting.

In a file

A project file stores a set as its semitones above C, such as `"0 3 6 9"`, which is easier to read and to write by hand than the number 585. The attribute `#[serde(try_from = "String", into = "String")]` saves it through a `String` and reads it back through `TryFrom<String>`, which refuses anything that is not a number from 0 to 11. The same pair of conversions, and why the reading one is `try_from`, is in [Twelve-tone rows and their rules](#).

Testing it

The tests in `pitch.rs` and `pitch_set.rs` read like the rules of this chapter. Names and octaves, at both ends of the range:

```
assert_eq!(Note::new(61).to_string(), "C#4");
assert_eq!(Note::new(61).name(Spelling::Flats), "Db4");
assert_eq!(Note::MIN.to_string(), "C-1");
assert_eq!(Note::MAX.to_string(), "G9");
```

Names with any letter, and the chromatic names:

```
assert_eq!(name(F, Letter::E), "E#");
assert_eq!(name(B, Letter::C), "Cb");
assert_eq!(name(G, Letter::F), "F##");
assert_eq!(name(A, Letter::B), "Bbb");
```

```
assert_eq!(chromatic.join(" "), "C C# D Eb E F F# G Ab A Bb B");
```

Intervals counted in letters, and the octave of a spelled note:

```
// A major third above A is C#, not Db.
assert_eq!(a.above(4, 2).to_string(), "C#");
```

```
assert_eq!(c4.name_as(NoteName::new(Letter::B, 1)), "B#3");
assert_eq!(Note::new(59).name_as(NoteName::new(Letter::C, -1)), "Cb4");
```

And reading, including what must not be read:

```
assert_eq!(read("F##dim7"), Some(("F##".to_string(), "dim7")));
assert_eq!(read("Ebbb"), Some(("Ebb".to_string(), "b")), "two at most");
assert_eq!(read("F#b"), Some(("F#".to_string(), "b")), "not mixed");
assert_eq!(read("c"), None, "lower case is a flat, not a note");
```

The sets' tests check that transposing wraps around (C major up 9 is A major, up 12 is itself) and that E minor seen from E is the set `0 3 7`.

Exercises on these types are in [Exercises: music theory in code](#).

Scales, modes and keys

Files: `crates/lineup-core/src/harmony/scale.rs`, `crates/lineup-core/src/key_map.rs`, `crates/lineup-core/src/tools.rs` and `crates/lineup-core/src/project.rs`

A key is what most of the app looks at before it shows anything: the piano roll lights up the key's rows, the harmony keys play its chords, the chord sheet names chords in it, the band plays in it, and every note's name depends on it. This chapter is about the `Scale` type that holds a key, how a key writes each of its notes, and what happens when the song changes key, all at once or partway through.

The music, briefly

A **scale** is a choice of some of the twelve pitch classes, counted up from a **root**. The major scale is the white keys from C, C D E F G A B: as semitones above the root, 0 2 4 5 7 9 11. The same pattern started on another note gives the major scale of that note: from D, it is D E F# G A B C#. A scale on a particular root, used as the home of a piece, is a **key**: D major.

The scales Ferrous Lineup knows, as semitones above the root:

Scale	Semitones
major	0 2 4 5 7 9 11
minor (natural)	0 2 3 5 7 8 10
harmonic minor	0 2 3 5 7 8 11
melodic minor	0 2 3 5 7 9 11
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
major pentatonic	0 2 4 7 9
minor pentatonic	0 3 5 7 10
blues	0 3 5 6 7 10
chromatic	all twelve

Harmonic minor raises the seventh note of the minor scale so that it leans on the root a semitone above it; melodic minor raises the sixth too. (Classical melodic minor falls back to the natural minor on the way down; the app uses the rising form both ways, as jazz musicians do.) The **modes**, Dorian to Locrian, are the major scale started on another of its notes: D Dorian is the white keys from D. So each mode shares its notes, and its key signature, with one major key: its **relative major**. A minor's relative major is C major, three semitones up.

The **key signature** is the sharps or flats written at the start of every line of a score. Each major key has its own, and going up a fifth adds a sharp: C major has none, G major one, D major two. Going down a fifth adds a flat: F major has one, B $\flat$ major two. That circle of keys a fifth apart is the **circle of fifths**. The manual's [Keys and modes](#) tells all of this with sounds to try.

Kinds of scale: an enum with data

```
#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash, Serialize, Deserialize)]
pub enum ScaleKind {
    Major,
    NaturalMinor,
    HarmonicMinor,
    MelodicMinor,
    Dorian,
    Phrygian,
    Lydian,
    Mixolydian,
    Locrian,
    MajorPentatonic,
    MinorPentatonic,
    Blues,
    Chromatic,
    Row(ToneRow),
    Set(PitchSet),
}
```

The thirteen scales of tonal music are plain variants. The last two carry their notes with them: a twelve-tone row and a pitch-class set can be the “key” of a song too, for music that has no key in the usual sense. They have chapters of their own ([Twelve-tone rows and their rules](#) and [Pitch-class sets, and one harmony with four names](#)); here they mostly show up as the cases that are different. One method sorts them out:

```
pub fn is_tonal(self) -> bool {
    !matches!(self, ScaleKind::Row(_) | ScaleKind::Set(_))
}
```

`Row(_)` matches a `Row` whatever row it holds.

The notes of a scale are listed as semitones above the root. For the tonal scales those lists never change, so they are written into the program and borrowed as `&'static [u8]`, slices that can have any length, five for a pentatonic scale, seven for the major scale:

```
fn tonal_intervals(self) -> &'static [u8] {
    match self {
        ScaleKind::Major => &[0, 2, 4, 5, 7, 9, 11],
        ScaleKind::NaturalMinor => &[0, 2, 3, 5, 7, 8, 10],
        // ...
        ScaleKind::Chromatic | ScaleKind::Row(_) | ScaleKind::Set(_) => {
            &[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
        }
    }
}
```

A row or a set has its own notes, which exist only inside the value, so nothing `'static` can point at them. The public `intervals` therefore returns an owned `Vec<u8>`, made from the row, the set or the table:

```
pub fn intervals(self) -> Vec<u8> {
    match self {
        ScaleKind::Row(row) => row.semitones().to_vec(),
        ScaleKind::Set(set) => set.semitones(),
        kind => kind.tonal_intervals().to_vec(),
    }
}
```

The last arm's pattern is a plain name, `kind`, which matches anything and binds it, so every tonal scale goes there.

A key: a root and a kind

```
#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash, Serialize, Deserialize)]
pub struct Scale {
    pub root: PitchClass,
    pub kind: ScaleKind,
    #[serde(default, skip_serializing_if = "std::ops::Not::not")]
    respelled: bool,
}
```

`respelled` is for the few keys that can be written two ways, below. Most of the methods on `Scale` are short iterator chains. The pitch classes of the scale are the root moved up by each interval:

```
pub fn pitch_classes(self) -> impl Iterator<Item = PitchClass> {
    self.kind
        .intervals()
        .into_iter()
        .map(move |interval| self.root.transpose(interval as i32))
}
```

`impl Iterator<Item = PitchClass>` says “some iterator of pitch classes” without naming its real type, which involves a closure, and closures have types nobody can write. `into_iter()` on the `Vec` gives its numbers by value, and `move` gives the closure its own copy of `self` (a `Scale` is `Copy`), since the iterator outlives the call.

The **degree** of a note is its place in the scale. `position` finds it, counting from 0, where musicians count from 1:

```
pub fn degree_of(self, pitch_class: PitchClass) -> Option<usize> {
    self.pitch_classes().position(|p| p == pitch_class)
}
```

In A minor, E is degree 4 (musicians say “the fifth”), and F $\sharp$ has no degree. `pitch_set` collects the same pitch classes into a `PitchSet`, so the test that every mode shares its notes with its relative major is a comparison of three numbers:

```
let c_major = Scale::default().pitch_set();
assert_eq!(Scale::new(D, ScaleKind::Dorian).pitch_set(), c_major);
assert_eq!(Scale::new(G, ScaleKind::Mixolydian).pitch_set(), c_major);
assert_eq!(Scale::new(A, ScaleKind::NaturalMinor).pitch_set(), c_major);
```

`Scale::default()` is C major, and `is_heptatonic` says whether a scale has seven notes. Chords stacked in thirds and Roman numerals only make sense in those, so many methods start by asking it ([Chords: building, naming and inversions](#)).

Sharps or flats: how a key is written

Every minor key and mode shares its key signature with its relative major, so the question “sharps or flats?” is answered by that major key. `relative_major` says how far up it is:

```
pub(crate) fn relative_major(self) -> u8 {
    match self {
        ScaleKind::Major
        | ScaleKind::MajorPentatonic
        | ScaleKind::Chromatic
        | ScaleKind::Row(_)
        | ScaleKind::Set(_) => 0,
        ScaleKind::NaturalMinor
        | ScaleKind::HarmonicMinor
        | ScaleKind::MelodicMinor
        | ScaleKind::MinorPentatonic
        | ScaleKind::Blues => 3,
        ScaleKind::Dorian => 10,
        ScaleKind::Phrygian => 8,
        ScaleKind::Lydian => 7,
        ScaleKind::Mixolydian => 5,
        ScaleKind::Locrian => 1,
    }
}
```

D Dorian is the white keys from D, and C is ten semitones above D; E Phrygian's C is eight above E, and so on. `pub(crate)` makes the method visible inside `lineup-core` only: the rest of the program asks the questions built on it.

The usual spelling then depends on that major key. The keys whose signature has flats are F, B $\flat$, E $\flat$, A $\flat$ and D $\flat$ major, and their minors and modes:

```
fn usual_spelling(self) -> Spelling {
    use PitchClass::*;
    match self.root.transpose(self.kind.relative_major() as i32) {
        F | ASharp | DSharp | GSharp | CSharp => Spelling::Flats,
        // F# major and Gb major are the same keys of the piano, and
        // the tonic's name decides between them. Their modes are
        // named with sharps (G# Dorian, B Lydian), except F Locrian,
        // whose notes are those of Gb major: F Gb Ab Bb Cb Db Eb.
        FSharp if self.root == F => Spelling::Flats,
        _ => Spelling::Sharps,
    }
}
```

The middle arm has a **match guard**, `if self.root == F`: the arm is taken only if the pattern matches and the condition holds. F Locrian is the one mode on those keys whose root is a white key with its own name; written with sharps it would be E $\sharp$ Locrian, so it takes the flats of G $\flat$ major.

The key signature, round the circle of fifths

A key signature is a number: how many sharps, or how many flats as a negative number. Going round the circle of fifths, the major key `n` fifths above C has `n` sharps. `key_signature` finds `n` by trying every step round the circle until it lands on the relative major:

```
pub fn key_signature(self) -> Option<(i8, bool)> {
    if !self.kind.is_tonal() || self.kind == ScaleKind::Chromatic {
        return None;
    }
    let major = self.root.transpose(self.kind.relative_major() as i32);
    // Going round the circle of fifths: each fifth up adds a sharp.
    let fifths = (0..12i8)
        .find(|&n| PitchClass::C.transpose(7 * i32::from(n)) == major)
        .unwrap_or(0);
    let sharps = match self.spelling() {
        Spelling::Flats => fifths - 12,
        Spelling::Sharps => fifths,
    };
    let minor = self.kind.relative_major() == 3;
    Some((sharps, minor))
}
```

Twelve fifths up come back to C, so a key with flats is the same number of fifths counted the other way: F major is eleven fifths up, which is one fifth down, so $11 - 12 = -1$, one flat. D $\flat$ major is seven fifths up and five down: five flats. The second value says whether the key is minor, which a MIDI file's key signature event stores too ([MIDI: messages, devices, recording and files](#)). A row, a set or the chromatic scale has no key signature, hence the `Option`.

The `0..12i8` range has its type written on the number, `12i8`, so `n` is an `i8` like the result. `find` stops at the first `n` that fits; one always does, and `unwrap_or(0)` only keeps the compiler from asking what happens if not.

Keys that can be written two ways

The keys with six sharps or six flats are the same keys of the piano: F $\sharp$ major and G $\flat$ major. Around them, C $\sharp$ major (seven sharps) is D $\flat$ major (five flats), and B major (five sharps) is C $\flat$ major (seven flats). Composers choose the name that fits the music around it; Debussy writes the middle of Golliwogg's Cakewalk in G $\flat$ major, a key of flats like the E $\flat$ major it comes from. Any other key would need eight signs or more the other way round: D $\sharp$ major needs nine sharps, so it is always E $\flat$ major.

So a `Scale` can be **respelled**: the same notes, the other name. `other_spelling` gives that key when there is one:

```
pub fn other_spelling(self) -> Option<Scale> {
    use PitchClass::*;
    let key = self.as_key();
    let major = key.root.transpose(key.kind.relative_major() as i32);
    let either_way = key.kind.is_tonal()
        && key.kind != ScaleKind::Chromatic
        && matches!(major, FSharp | CSharp | B);
    either_way.then_some(Scale {
        respelled: !self.respelled,
        ..self
    })
}
```

`Scale { respelled: !self.respelled, ..self }` is **struct update syntax**: a copy of `self` with one field changed. `spelling()` turns the usual answer round when the key is respelled, and the tonic takes the next letter: F♯ becomes G♭, B becomes C♭.

The field is private, so the only ways to set it are the methods, which check that the key can be written the other way. `Scale::written(tonic, kind)` builds a key from how its tonic is written, which is what the Key menus and the key line of a typed chord sheet use: G♭ with “major” gives the respelled F♯ major, and G♭ with “minor”, a key that would need a double flat in its signature, gives the usual F♯ minor.

In a file, the attribute `#[serde(default, skip_serializing_if = "std::ops::Not::not")]` leaves the field out when it is `false`, which is almost always, and reads a missing one as `false`. `skip_serializing_if` takes the path of a function that receives `&bool` and returns `true` to skip; `std::ops::Not::not` is the `!` operator as a function, so a field is skipped when `!respelled`. A file saved with a respelled key and opened by a version of the app that does not know the field gets the key under its usual name, with the same notes.

A letter for every note

`spelling()` only says how the black keys of the tonic are written. A key’s notes need more: D harmonic minor has a B♭ and a C♯, and F♯ major has an E♯, which is neither the sharp nor the flat name of any black key. The rule is the one from [Pitch classes, notes and spelling](#): a seven-note scale uses each letter once. `spell` gives a pitch class the letter of its place in the scale, counted up from the tonic’s letter, and lets `NoteName::with_letter` find the sharps or flats:

```
pub fn spell(self, pitch_class: PitchClass) -> NoteName {
    if let Some(scale) = self.set_as_scale() {
        return scale.spell(pitch_class);
    }
    if !self.kind.is_tonal() {
        return NoteName::chromatic(pitch_class);
    }
    let name = NoteName::with_letter(
        pitch_class,
        self.tonic_name()
            .letter
            .up(i32::from(self.letters_above_tonic(pitch_class))),
    );
    if self.has_letter_for(pitch_class) {
        name
    } else {
        name.simplified()
    }
}
```

How many letters above the tonic a note goes is its place in the scale for a seven-note scale. The pentatonic and blues scales leave letters out, so they have a table of their own:

```
fn letter_steps(self) -> &'static [u8] {
    match self {
        ScaleKind::MajorPentatonic => &[0, 1, 2, 4, 5],
        ScaleKind::MinorPentatonic => &[0, 2, 3, 4, 6],
        ScaleKind::Blues => &[0, 2, 3, 4, 4, 6],
        ScaleKind::Chromatic | ScaleKind::Row(_) | ScaleKind::Set(_) =>
&CHROMATIC_LETTERS,
        _ => &[0, 1, 2, 3, 4, 5, 6],
    }
}
```

The blues scale's two fourth-letter entries are its "blue note", a flat fifth next to the fifth: A blues is A C D E_b E G, not A C D D_# E G.

Notes from outside the key

A note outside the scale has no place in it. It is written the way musicians most often meet it, counted from the tonic: a flat second, third, sixth or seventh, a sharp fourth, and the major sixth and seventh, which in a minor key are the raised notes of melodic and harmonic minor. That is a table of letters by semitones above the tonic:

```
const CHROMATIC_LETTERS: [u8; 12] = [0, 1, 1, 2, 2, 3, 3, 4, 5, 5, 6, 6];
```

One semitone above the tonic is one letter up (a flat second), six semitones is three letters up (a sharp fourth), eleven is six letters up (the major seventh). So in C major the black keys outside the key are D $\flat$, E $\flat$, F $\sharp$, A $\flat$ and B $\flat$, and in D minor C $\sharp$ is C $\sharp$.

`letters_above_tonic` chooses between the scale's place and this table:

```
fn letters_above_tonic(self, pitch_class: PitchClass) -> u8 {
    let semitones = self.root.semitones_up_to(pitch_class);
    if self.has_letter_for(pitch_class)
        && let Some(place) = self
            .kind
            .tonal_intervals()
            .iter()
            .position(|&interval| interval == semitones)
    {
        return self.kind.letter_steps()[place];
    }
    CHROMATIC_LETTERS[semitones as usize]
}
```

The `if` joins a condition and a `let` with `&&`. This is a **let chain**: the block runs only if the condition is true and the pattern matches, and `place` is bound inside it. Without it, the code would need an `if` inside an `if`, or a `match` on a tuple.

The table can ask for two signs. In D $\flat$ major the note a semitone above the tonic, D, is a flat second, the letter E: E $\flat$. No musician writes that for a note outside the key, so `spell` takes the simpler name, D, for any note the scale has no letter for. Inside the scale two signs stay, because the key needs them: G $\sharp$ harmonic minor's seventh note is F $\times$.

A pitch-class set whose notes are exactly a tonal scale on its root is spelled as that scale (`set_as_scale`), so E major shown as a set keeps its G $\sharp$, C $\sharp$ and D $\sharp$. A row or any other set has no letters, so its notes take the chromatic names.

Names with octaves

`note_name` puts the spelled name together with the right octave number, through `Note::name_as`:

```
pub fn note_name(self, note: Note) -> String {
    note.name_as(self.spell(note.pitch_class()))
}
```

C $\sharp$ harmonic minor raises its seventh, B, to B $\sharp$, so in that key MIDI note 60, middle C, is "B $\sharp$ 3". The voice-leading check and the part motion view name the notes they talk

about this way ([Voice leading and part motion](#)).

Changing key: moving the song

The **Key** menus, left of the harmony keys and at the top of the piano roll, set the key of the whole song, the one the harmony keys, the hints and the band use. So they move the song too, the way a band plays the same song in another key when asked: pick D and the chords, the bass and the melody all go up a tone; pick minor and every note moves to the same place in the minor scale. The drums stay as they are, and **Ctrl+Z** puts everything back in one go.

As with every [editing tool](#), the musical part is a plain function that changes nothing itself. It answers one question: where does this note go when the key changes from `from` to `to`?

```

pub fn change_key(note: Note, from: Scale, to: Scale) -> Note {
    let from = from_key(from, to);
    if !from.kind.notes_follow(to.kind) {
        return note;
    }
    let up = from.root.semitones_up_to(to.root) as i32;
    let shift = if up > 6 { up - 12 } else { up };
    let Some(moved) = note.transpose(shift) else {
        return note;
    };
    // The chromatic scale holds every note, so it has no degrees to
    // match with a key's: into it or out of it, notes only take the new
    // root. (Out of it, every note would otherwise be "of the old scale"
    // and be pulled into the new key, losing every sharp and flat.)
    if from.kind == ScaleKind::Chromatic || to.kind ==
ScaleKind::Chromatic {
        return moved;
    }
    // The old scale moved to the new root, so `moved` can be compared
    // with it degree by degree.
    let old = Scale::new(to.root, from.kind);
    let Some(degree) = old.degree_of(moved.pitch_class()) else {
        return moved;
    };
    let (old_steps, new_steps) = (from.kind.intervals(),
to.kind.intervals());
    if old_steps.len() == new_steps.len() {
        let offset = new_steps[degree] as i32 - old_steps[degree] as i32;
        moved.transpose(offset).unwrap_or(moved)
    } else {
        nearest_in_key(moved, to)
    }
}

```

It works in two moves.

1. **A new root moves every note by the same distance.** C to D is two semitones up. C to G could be seven up or five down; `shift` takes the nearer one, so the song stays roughly where it was on the keyboard instead of jumping up a fifth.
2. **A new mode moves each note to the same degree.** After the first move the note is compared with the *old* kind of scale on the *new* root. If it is the third note of D major (F $\sharp$), it becomes the third note of D minor (F). The difference between the two interval lists at the same place is how far that degree moves:

degree	1	2	3	4	5	6	7
major	0	2	4	5	7	9	11
natural minor	0	2	3	5	7	8	10
moves by	0	0	-1	0	0	-1	-1

So C major to C minor lowers E, A and B, and leaves C, D, F and G.

Some notes need a rule of their own:

- A **note outside the old scale**, such as a note of a borrowed chord or of a part written in another key, has no degree, so the `let ... else` returns it after the first move only. It keeps its distance from the root, and a modulation stays a modulation.
- **Scales of different sizes**, like major (seven notes) and major pentatonic (five), have no matching degrees. A note that is not in the new scale goes to the nearest one that is, with `nearest_in_key` ([Editing tools as plain functions](#)).
- The **chromatic scale** has every note, so going into it or out of it only moves the root.
- Between a key and a **row** or a **set**, there is nothing to move by, and `notes_follow` says so: only what the piano roll highlights changes. A set whose notes are exactly a key, such as C major shown as the set 7-35, moves on as that key (`from_key`).

`let Some(moved) = ... else { return note; };` is a **let-else**: if the pattern does not match, the `else` block must leave the function. Here it covers “the note would leave the MIDI range” and “the note is not in the scale”.

The whole project

`Project::change_key` runs that function over every note that should move. Its core loop:

```

let pitched: Vec<bool> =
self.channels.iter().map(Channel::is_pitched).collect();
for (index, pattern) in self.patterns.iter_mut().enumerate() {
    let (old, new) = (before[index], after[index]);
    for (channel, _) in pitched.iter().enumerate().filter(|(_, p)| **p) {
        if pattern.is_step_row(channel) {
            continue;
        }
        let notes: Vec<NoteEvent> = pattern
            .notes(channel)
            .iter()
            .map(|note| NoteEvent {
                pitch: tools::change_key(note.pitch, old, new),
                ..*note
            })
            .collect();
        pattern.set_notes(channel, notes);
    }
}

```

- `pitched` is collected before the loop because the loop borrows `self.patterns` mutably, and Rust will not let the code look at `self.channels` through `self` at the same time. A small `Vec<bool>` read first ends that borrow early.
- `Channel::is_pitched` is false for the built-in drums and for a MIDI Out channel on the General MIDI drum channel: drums do not move.
- A row of step-sequencer steps (every note a one-step C4) is a sound played on the beat, not a tune, so it stays as it is.
- `before[index]` and `after[index]` are the keys each pattern is played in before and after the change, which are not always the song's key, as the next section shows.

The app calls it through `edit`, which keeps the project as it was before ([Undo and redo](#)), so one **Ctrl+Z** restores the key and every note:

```

self.edit("change key", |project| project.change_key(scale));

```

The chord sheet and the band's parts move with the notes; they are in [The chord sheet: model, layout and text](#) and [How the band plays a chord sheet](#).

Key changes partway through: key markers

Many pieces do not stay in one key. Mozart's Rondo alla Turca starts in A minor, turns to A major at bar 25, where the key signature gets three sharps, and goes back to A minor at bar 65. Moving to a new key is a **modulation**. Three sharps are also the

signature of F# minor, A major's relative minor, and the A major part passes through F# minor without changing its signature; you hear which key the music is in by where it comes home, not by the signature.

A key change is one more marker on the playlist, next to the time signature and tempo markers:

```
pub struct KeyMarker {  
    /// The step it starts on, counted from the start of the song.  
    pub start: usize,  
    pub scale: Scale,  
}
```

A key marker never moves a note. It changes what the key is called, which rows the piano roll lights up, which chords the harmony keys offer and which chords count as “in the key”. That is the difference from the Key menus, which move the song.

`KeyMap` answers “what key is step 300 in?": the song's own key until the first marker, then each marker's key in turn. Building one puts the markers in order and keeps one per step:

```
pub fn new(first: Scale, markers: impl IntoIterator<Item = KeyMarker>) ->  
Self {  
    let mut markers: Vec<KeyMarker> = markers  
        .into_iter()  
        .filter(|marker| marker.start > 0)  
        .collect();  
    markers.sort_by_key(|marker| marker.start);  
    markers.dedup_by_key(|marker| marker.start);  
    KeyMap { first, markers }  
}
```

`dedup_by_key` removes an item when it has the same key as the one before it, so after sorting it keeps the first marker on each step. Looking a step up takes the last marker that starts at or before it:

```
pub fn marker_at(&self, step: usize) -> Option<&KeyMarker> {  
    self.markers  
        .iter()  
        .take_while(|marker| marker.start <= step)  
        .last()  
}
```

`take_while` stops at the first marker after `step`, and `last` gives the one before it, or `None` if the step comes before every marker. A `Project` builds its map with `key_map()`, and `key_at(step)` asks it.

Adding a marker keeps the list sorted with `binary_search_by_key`, which gives `Ok(index)` if a marker already starts on that step and `Err(index)`, where it would go, if not:

```
let markers = &mut self.playlist.key_markers;
match markers.binary_search_by_key(&marker.start, |m| m.start) {
    Ok(index) => markers[index] = marker,
    Err(index) => markers.insert(index, marker),
}
```

A marker on step 0 is not added at all: it changes the key the song starts in.

The key of a pattern

A pattern is written once and played wherever its clips are, and a song can play it in different keys. The piano roll needs one key to light up its notes, so a pattern takes the key where its first clip starts:

```
pub fn pattern_key_at(&self, pattern: usize, step: usize) -> Scale {
    self.playlist
        .clips
        .iter()
        .filter(|clip| clip.pattern == pattern)
        .map(|clip| clip.start)
        .min()
        .map_or(self.scale, |start| self.key_at(start + step))
}
```

`pattern_key(p)` is this at step 0. The piano roll says *This pattern: A major* beside the Key menus, and the harmony keys say *Now in A major* when the playhead is in a key that is not the song's own.

When the song moves, the markers move

When the Key menus move a song with key changes, every key change must move by the same interval, or the march of the Rondo would stay in A major while the rest went to C minor. `moved_key` moves a key the way the song's key moved:

```
pub fn moved_key(key: Scale, from: Scale, to: Scale) -> Scale {
    let from = crate::tools::from_key(from, to);
    if !from.kind.notes_follow(to.kind) {
        return key;
    }
    let up = from.root.semitones_up_to(to.root);
    let root = key.root.transpose(up as i32);
    let kind = if key.kind == from.kind && from.kind.is_tonal() &&
to.kind.is_tonal() {
        to.kind
    } else {
        key.kind
    };
    if root == key.root && kind == key.kind {
        return key;
    }
    Scale::new(root, kind)
}
```

The root moves by the same semitones. A key in the song's old mode takes the new mode: a song in C major with a passage in G major, moved to C Dorian, has that passage in G Dorian. A key in another mode keeps its own: in a song in A minor moved up a minor third to C minor, the A major march becomes C major and the F# minor episode A minor, as a transposed score would read. Then `change_key` moves each pattern from the key it was played in to the key it is played in now, which is what `before` and `after` held above.

MIDI files store key changes too, as key signature events; see [MIDI: messages, devices, recording and files](#). A marker can also be written by the app itself, from the keys it hears in the notes ([Hearing harmony: chords and keys from notes](#)).

Snap to scale

The piano roll's **Snap to scale** button makes a note you draw or drag land on the nearest note of the key, and its right-click menu adds **Snap notes you play**, for the typing keyboard and a MIDI keyboard. Both come down to

`tools::nearest_in_key(note, scale)`, in [Editing tools as plain functions](#); the key it snaps to is the key at that point of the song, after key markers. How a MIDI keyboard's keys are snapped on the MIDI thread is in [MIDI: messages, devices, recording and files](#).

Testing it

The tests in `scale.rs` state the rules for every key at once, not just for the keys someone thought of. Every seven-note scale, on every root, written either way, uses seven different letters, and each name still names the right key of the piano:

```
let mut letters: Vec<_> = scale
    .pitch_classes()
    .map(|p| scale.spell(p).letter)
    .collect();
letters.sort();
letters.dedup();
assert_eq!(letters.len(), 7, "{}", scale.name());
```

The signs a key's notes are written with add up to its key signature, for every mode on every root, so a staff never needs an accidental for a note of the key:

```
let (signature, _) = scale.key_signature().unwrap();
let written: i8 = scale
    .pitch_classes()
    .map(|note| scale.spell(note).accidental)
    .sum();
assert_eq!(written, signature, "{}", scale.name());
```

And the signatures of all twelve major keys, from C upwards, are the circle of fifths:

```
// Sharps count up, flats count down, round the circle of fifths:
// C, Db, D, Eb, E, F, F#, G, Ab, A, Bb, B.
let major = [0, -5, 2, -3, 4, -1, 6, 1, -4, 3, -2, 5];
```

`tools.rs` checks the moves of a key change: C to G goes down a fourth, C major to C minor lowers E, A and B, a note outside the scale only takes the new root, and a D7 taken to the chromatic scale and back is still a D7. `key_map.rs` checks that the Rondo's march and episode land in C major and A minor when the song goes up a minor third.

Exercises on keys are in [Exercises: music theory in code](#).

Chords: building, naming and inversions

Files: `crates/lineup-core/src/harmony/chord.rs`, `crates/lineup-core/src/harmony/symbol.rs`, `crates/lineup-core/src/harmony/analysis.rs` and `crates/lineup-core/src/harmony/scale.rs`

Chords are everywhere in Ferrous Lineup: the piano roll places them with one click and names whatever is sounding, the chord sheet reads them as you type them, the harmony keys play them, and the band turns them into parts. This chapter is about the types behind all of that: a chord's quality, a chord, a chord symbol as a lead sheet writes it, and the analysis that names a group of notes.

The music, briefly

A **chord** is several notes played together. Most chords are built from **thirds**: a major third is four semitones, a minor third three.

Chord	Built from the root	On C	Sounds
major	major third + minor third (0 4 7)	C E G	bright, settled
minor	minor third + major third (0 3 7)	C E $\flat$ G	darker
diminished	two minor thirds (0 3 6)	C E $\flat$ G $\flat$	tense
augmented	two major thirds (0 4 8)	C E G $\sharp$	unstable, dreamy
dominant seventh	major triad + minor third (0 4 7 10)	C E G B $\flat$	wants to move on
major seventh	major triad + major third (0 4 7 11)	C E G B	soft, jazzy
minor seventh	minor triad + minor third (0 3 7 10)	C E $\flat$ G B $\flat$	mellow

Going on up in thirds past the octave gives the **extended** chords of jazz, soul and funk. Their names count the notes of the scale from the root: the ninth is the second an octave up ($14 = 12 + 2$ semitones), the eleventh the fourth ($17 = 12 + 5$), the thirteenth the sixth ($21 = 12 + 9$). C13 is C E G B $\flat$ D A.

The lowest note is the **root** when the chord is in **root position**. Played as E G C, with E at the bottom, it is still C major, in its first **inversion**. A **chord symbol**, as a lead sheet writes it, puts the bass after a slash: C/E. The manual's [Chords and their names](#) has all of this, with what to listen for.

Qualities: a table in a match

A chord without its root is a **quality**: “minor seventh”, whatever note it starts on. There are thirty, one enum variant each:

```
#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash)]
pub enum ChordQuality {
    Major,
    Minor,
    Diminished,
    Augmented,
    Sus2,
    Sus4,
    Major6,
    Minor6,
    Dominant7,
    Major7,
    Minor7,
    // ...
    Dominant13,
    Major13,
}
```

Everything Ferrous Lineup knows about a quality is a `match` on it, one method per fact. The notes, as semitones above the root:

```
pub fn intervals(self) -> &'static [u8] {
    use ChordQuality::*;
    match self {
        Major => &[0, 4, 7],
        Minor => &[0, 3, 7],
        Diminished => &[0, 3, 6],
        // ...
        Dominant9 => &[0, 4, 7, 10, 14],
        // ...
        // Players leave the eleventh out of thirteenth chords, because
        // it clashes with the third; so does Ferrous Lineup.
        Dominant13 => &[0, 4, 7, 10, 14, 21],
        Major13 => &[0, 4, 7, 11, 14, 21],
    }
}
```

The ninth is written as 14, not 2. As a pitch class it is the same, but a chord placed in the piano roll is built from these numbers, and a ninth a whole tone above the root would sound like a cluster, not a ninth chord.

The other facts are methods of the same shape: `symbol` (the “m7” of “Am7”), `name` (“minor seventh”), `tones` (which note of the chord each interval is, below), `is_minor`, `is_dominant` and `numeral_suffix` (the “°” of “vii°”). Writing knowledge as `match` arms has one big advantage over, say, a list of structs: add a variant and the compiler names every method that does not handle it yet.

`numeral_suffix` ends with a **binding pattern**:

```
match self {
  Major | Minor => "",
  Diminished => "°",
  // ...
  other => other.symbol(),
}
```

`other` matches anything left, like `_`, but gives it a name to use. `family`, which sorts a quality into the three kinds of triad, adds a guard to one:

```
pub fn family(self) -> ChordFamily {
  use ChordQuality::*;
  match self {
    Diminished | HalfDiminished7 | Diminished7 =>
      ChordFamily::Diminished,
    quality if quality.is_minor() => ChordFamily::Minor,
    _ => ChordFamily::Major,
  }
}
```

Keys use the family to match a chord to their own chords: G7 is the major chord on G, Bm7b5 the diminished one on B ([Roman numerals, the harmony keys and chord functions](#)).

Which quality is this?

A quality’s notes on C are a `PitchSet`, and finding the quality of a group of notes seen from its root is one search:

```
pub fn identify(relative: PitchSet) -> Option<ChordQuality> {
    Self::ALL
        .into_iter()
        .find(|quality| quality.pitch_set() == relative)
}
```

`find` returns the first quality that fits. So the order of `ChordQuality::ALL` matters, and it lists the simpler, more common chords first. For `identify` to find every quality, no two may have the same notes, and a test makes sure of it:

```
#[test]
fn no_two_qualities_have_the_same_notes() {
    // Otherwise `identify` could never find the second one.
    for (i, a) in ChordQuality::ALL.iter().enumerate() {
        for b in &ChordQuality::ALL[i + 1..] {
            assert_ne!(a.pitch_set(), b.pitch_set(), "{a:?} and {b:?}");
        }
    }
}
```

`&ChordQuality::ALL[i + 1..]` is the slice of the array from `i + 1` to the end, so every pair is compared once.

A chord: a root and a quality

```
#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash)]
pub struct Chord {
    pub root: PitchClass,
    pub quality: ChordQuality,
}
```

Its pitch classes are the quality's moved up to the root, and its notes are stacked up from a given root note, leaving out any that would fall off the top of MIDI:

```
pub fn pitch_set(self) -> PitchSet {
    self.quality.pitch_set().transpose(self.root.semitone())
}

pub fn notes_from(self, root_note: Note) -> Vec<Note> {
    self.quality
        .intervals()
        .iter()
        .filter_map(|&interval| root_note.transpose(interval as i32))
        .collect()
}
```

The chords of a key: stacking thirds

To build the chord on a note of a key, take every other note of the scale: the note, the one two places up, the one four places up (and six places up for a seventh chord). This is **stacking thirds**. In C major, starting on D gives D, F and A: D minor.

```
pub fn stack_thirds(self, note: Note, count: usize) -> Option<Vec<Note>> {
    if !self.is_heptatonic() {
        return None;
    }
    let intervals = self.kind.tonal_intervals();
    let degree = self.degree_of(note.pitch_class())?;
    let notes = (0..count)
        .map(|i| {
            let index = degree + 2 * i;
            // Going past the seventh note wraps into the next octave.
            let octave = (index / 7) as i32;
            let above_root = intervals[index % 7] as i32 + 12 * octave;
            above_root - intervals[degree] as i32
        })
        .filter_map(|semitones| note.transpose(semitones))
        .collect();
    Some(notes)
}
```

Read it as a pipeline: the numbers `0..count` become places in the scale, the places become semitones above `note`, and `filter_map` turns the semitones into notes, dropping any that would leave the MIDI range. The `?` after `degree_of` returns `None` early when the note is not in the scale.

`Scale::triad(degree)` and `seventh(degree)` stack three or four notes on a degree and name the result with `identify`. For all seven degrees of C major, that is the table every songwriter learns:

Degree	1	2	3	4	5	6	7
Triad	C	Dm	Em	F	G	Am	Bdim
Numeral	I	ii	iii	IV	V	vi	vii°
Seventh	Cmaj7	Dm7	Em7	Fmaj7	G7	Am7	Bm7b5

Change the key and the table follows, because it is computed, not written down.

`ninth(degree)` stacks five notes, and returns `None` on the degrees where the ninth is only a semitone above the root an octave up (E G B D F in C major), which clashes, and no named chord fits.

Chord stamps

What a click in the piano roll places is a **stamp**, an enum whose variants carry what they need:

```
pub fn notes_at(self, note: Note, scale: Scale) -> Vec<Note> {
  match self {
    ChordStamp::Single => vec![note],
    ChordStamp::ScaleTriad => scale.stack_thirds(note,
3).unwrap_or_else(|| vec![note]),
    ChordStamp::ScaleSeventh => scale.stack_thirds(note,
4).unwrap_or_else(|| vec![note]),
    ChordStamp::Fixed(quality) => Chord::new(note.pitch_class(),
quality).notes_from(note),
    ChordStamp::Set(set) => set
      .semitones()
      .into_iter()
      .filter_map(|up| note.transpose(i32::from(up)))
      .collect(),
  }
}
```

The “Chord in key” stamps are where the piano roll teaches harmony: click any lit row and you get the chord the key gives you on that note, without knowing its name first. The name then appears in the chord lane above the grid. A `Set` stamp places any group of notes picked in the set browser ([Pitch-class sets, and one harmony with four names](#)).

The **Draw** menu lists every stamp. `all()` chains two iterators: the three special stamps, then one `Fixed` stamp per quality:

```
pub fn all() -> impl Iterator<Item = ChordStamp> {
  [
    ChordStamp::Single,
    ChordStamp::ScaleTriad,
    ChordStamp::ScaleSeventh,
  ]
  .into_iter()
  .chain(ChordQuality::ALL.into_iter().map(ChordStamp::Fixed))
}
```

`map(ChordStamp::Fixed)` passes the variant itself as a function: a tuple variant like `Fixed` is a function from `ChordQuality` to `ChordStamp`.

Next to the Draw menu, a second menu picks the **inversion** of a stamped chord. Inverting moves the lowest note up an octave (`tools::inversion`, in [Editing tools as plain functions](#)), which would move the chord away from where you clicked, so the piano roll moves it back down an octave and the note you clicked stays the root:

```
fn stamp_notes(&self, pitch: Note, scale: Scale) -> Vec<Note> {
    let notes = tools::inversion(&self.stamp.notes_at(pitch, scale),
self.inversion);
    if notes.contains(&pitch) {
        return notes;
    }
    notes
        .iter()
        .map(|note| note.transpose(-12))
        .collect::<Option<Vec<Note>>>()
        .unwrap_or(notes)
}
```

Collecting an iterator of `Option`s into an `Option<Vec<_>>` gives `None` as soon as one item is `None`. Here the type is written with a **turbofish**, `collect::<...>()`, because `unwrap_or` is called on the result straight away, and Rust must know the result's type to find that method. If a note would fall off the bottom of MIDI, the chord stays as it was. The outline that follows the pointer is drawn with the same function, so what you see is what a click places.

Spelling a chord

C#7 and D $\flat$ 7 are played on the same keys, but written differently. A chord is spelled up from its root by letters: the third is two letters up, the fifth four, the seventh six.

Chord	Root	Third	Fifth	Seventh
C#7	C#	E#	G#	B
D $\flat$ 7	D $\flat$	F	A $\flat$	C $\flat$

The seventh of D $\flat$ 7 is C $\flat$, not B: six letters above D is C. So each quality also says which note of the chord each interval is, counted the way musicians count:

```
pub fn tones(self) -> &'static [u8] {
    use ChordQuality::*;
    match self {
        Major | Minor | Diminished | Augmented => &[1, 3, 5],
        Sus2 => &[1, 2, 5],
        Sus4 => &[1, 4, 5],
        // ...
        Dominant13 | Major13 => &[1, 3, 5, 7, 9, 13],
    }
}
```

and `spell` walks the intervals and the tones together with `zip`, asking `NoteName::above` for each note: so many semitones, `tone - 1` letters up:

```
pub fn spell(self, root: NoteName) -> Vec<NoteName> {
    self.quality
        .intervals()
        .iter()
        .zip(self.quality.tones())
        .map(|(&interval, &tone)| root.above(interval % 12, tone - 1))
        .collect()
}
```

That gives A major as A C# E, C diminished seventh as C Eb Gb Bbb (its seventh is a seventh, even though it sounds like a sixth), and E7#9 as E G# B D Fx: the sharp ninth is a ninth, one letter above the root, so it is a kind of F, not G.

A name for the root

Spelling needs the root's name to start from. In a key, the key gives it:

`Scale::spell_root` writes a root in the scale as the scale writes it, and a root from outside the scale from the tonic, as `Scale::spell` does (Eb major in C major, the chord borrowed from C minor). One more rule keeps chord names readable: if that root would spell the chord with two sharps or flats on one note, the other name of the root is used. In Db major, the chord a semitone above the tonic is written D major, not Eb major.

```
let doubled = |root: NoteName| chord.spell(root).iter().any(|n|
n.accidental.abs() >= 2);
if !doubled(strict) {
    return strict;
}
let other = chord.root_name();
if doubled(other) {
    strict.simplified()
} else {
    other
}
```

Without a key, as for a twelve-tone row, `Chord::root_name` picks the sharp or the flat name of the root, whichever spells the whole chord with fewer signs. C# major is C# E# G#, three sharps; Db major is Db F Ab, two flats; so it is Db.


```

let usual = NoteName::chromatic(self.root);
[Spelling::Sharps, Spelling::Flats]
  .into_iter()
  .map(|spelling| NoteName::plain(self.root, spelling))
  .fold(usual, |best, other| {
    if accidentals(other) < accidentals(best) {
      other
    } else {
      best
    }
  })
})

```

`fold` carries a value through the iterator: it starts with `usual` and, for each item, the closure returns the value to carry on. Here it keeps whichever name has fewer signs, and on a tie the one it already has, the usual name of the black key.

`Scale::chord_name(chord)` and `Scale::chord_notes(chord)` put it together: the chord on the third note of A major is “C#m”, and in D minor A major’s notes are A C# E.

Chord symbols: reading a lead sheet

A song you want the band to play usually comes as a **lead sheet**, and a lead sheet names its chords: “C | G/B | Am | Am/G”. The chord sheet reads exactly that ([The chord sheet: model, layout and text](#)), through `ChordSymbol`:

```

#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash)]
pub struct ChordSymbol {
    /// The root, spelled as written: Db7 keeps its Db, it is not C#7.
    pub root: NoteName,
    pub quality: ChordQuality,
    /// The note under the chord, after the slash: the B of "G/B". `None`
    /// when the root is in the bass, as usual.
    pub bass: Option<NoteName>,
}

```

A symbol keeps its root as written, a `NoteName`, so a chart in D_b stays in D_b. To play it, `chord()` turns it into a `Chord`, a pitch class and a quality.

Many ways to write one chord

Musicians write the same quality in several ways: a minor seventh is Cm7, Cmin7 or C-7; a major seventh Cmaj7, CM7 or CΔ7; a half-diminished chord Cm7b5 or Cø. All of them are read, from a table of tuples:

```
const SPELLINGS: [(ChordQuality, &[&str]); 30] = {
    use ChordQuality::*;
    [
        (Major, &["", "maj", "M"]),
        (Minor, &["m", "min", "mi", "-"]),
        (Diminished, &["dim", "°", "o"]),
        // ...
        (Major13, &["maj13", "M13", "Δ13"]),
    ]
};
```

Each entry pairs a quality with a slice of strings, `&[&str]`. Slices can have different lengths, so one quality can have one spelling and another six in the same array. The block `{ use ChordQuality::*; [...] }` is an expression whose value is its last line, so the `use` reaches only the table. Reading a quality cleans the text and searches:

```
fn read_quality(text: &str) -> Option<ChordQuality> {
    let plain: String = text
        .chars()
        .filter(|c| !matches!(c, '(' | ')' | ' '))
        .map(|c| match c {
            'b' => 'b',
            '#' => '#',
            other => other,
        })
        .collect();
    SPELLINGS
        .iter()
        .find(|(_, spellings)| spellings.contains(&plain.as_str()))
        .map(|(quality, _)| *quality)
}
```

Brackets and spaces are dropped, so “C7(b9)” reads like “C7b9”, and `b` and `#` work like `b` and `#`. Upper and lower case matter in one place, an important one: CM7 is a major seventh (C E G B) and Cm7 a minor seventh (C E_b G B_b). When Ferrous Lineup writes a chord back, it uses one way for each, the one `symbol` gives: Cmaj7, Cm7, Cm7b5, Cdim7. A test writes every quality on every root, with and without a bass, and reads it back, so the table and `symbol` cannot drift apart.

Slash chords, and the six-nine chord

A slash puts another note in the bass. There are two kinds:

- **Inversions:** the bass note is in the chord. C/E is C major with its third in the bass, C/G with its fifth, G7/F with the seventh of G7. `ChordSymbol::bass_in_chord` says which: 3, 5 or 7.

- **A chord over a bass note:** the bass is not in the chord. F/G is F major over a G, an open sound that pop and gospel players often use instead of G11.

`bass_in_chord` says `None`.

One slash is not a bass note: the one in C6/9, the six-nine chord. So only what follows the **last** slash, and only if it is a note name, is a bass:

```
fn split_bass(rest: &str) -> (&str, Option<NoteName>) {
    match rest.rsplit_once('/') {
        Some((before, after)) => match NoteName::parse(after.trim()) {
            Some(bass) => (before, Some(bass)),
            None => (rest, None),
        },
        None => (rest, None),
    }
}
```

`rsplit_once('/')` splits the text at its last slash into the parts before and after it, or gives `None` when there is none. “C6/9” is a six-nine chord, and “C6/9/E” a six-nine chord over E.

Reading a whole symbol

`ChordSymbol::parse` reads the root with `NoteName::read_either`, in letters or in solfège, keeping the reading whose rest is a chord, so “Faug” is F augmented and “Lam7” is A minor seventh:

```
let read = NoteName::read_either(text, |rest| read_rest(rest).is_some());
let Some((root, rest)) = read else {
    return Err(match NoteName::read_either(text, |_| true) {
        None => trf(
            "\"{}\" does not start with a note: chords start with a letter
from A \
            to G, like C, F#m or Bb7, or with a syllable, like Do, Fa#m
or Sib7.",
            &[&text],
        ),
        Some((_, rest)) => trf(
            "\"{}\": \"{}\" is not a kind of chord Ferrous Lineup knows, \
            such as m, 7, maj7, m7b5, sus4 or 13.",
            &[&text, &split_bass(rest).0],
        ),
    });
};
```

The error says what went wrong, so the chord sheet can show it next to the bar where it was typed: either the text does not start with a note, or what follows the note is no

chord Ferrous Lineup knows.

`ChordSymbol` also implements the standard trait `FromStr`, “can be read from text”, by calling `parse`. With it, `str::parse` works for chords as it does for numbers:

```
impl FromStr for ChordSymbol {
    type Err = String;

    fn from_str(text: &str) -> Result<Self, Self::Err> {
        ChordSymbol::parse(text)
    }
}
```

so `let chord: ChordSymbol = "Am7".parse()?;` reads a chord.

Moving a chart to another key

To move a song to another key, a musician moves every chord by the same distance, counted in letters as well as semitones. From E minor to D minor every letter goes down one:

E minor	F#m7b5	B7	Em
D minor	Em7b5	A7	Dm
C minor	Dm7b5	G7	Cm
B♭ minor	Cm7b5	F7	B♭m
G# minor	A#m7b5	D#7	G#m

This is a **ii-V-i**, the most common way to arrive in a minor key: in every key the ii is one letter above the tonic and the V four letters above. Counting by letters keeps it that way: in G# minor the V is D#7, never E♭7. `transposed(from, to)` does it, moving the root and the bass with `NoteName::above` and taking the simpler name for a root that would need two signs, since chord symbols never have them ([Pitch classes, notes and spelling](#)). A test checks the ii-V-i in all twelve keys, written with sharps and with flats.

Naming what is playing

The piano roll names whatever sounds at each step: nothing, one note, two notes, a chord it knows, or a group with no chord name. Each answer needs different data, which is a job for an enum whose variants carry data:

```

#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub enum Harmony {
    /// No notes at all.
    Silence,
    /// One pitch class, maybe in several octaves.
    Single(PitchClass),
    /// Two pitch classes. `low` is the one in the bass, and `semitones`
    /// counts up from it to the other.
    Interval {
        low: PitchClass,
        high: PitchClass,
        semitones: u8,
    },
    /// A chord we know. When `bass` is not the chord's root, the chord is
    /// an inversion, written with a slash: C/E is C major with E in the
    bass.
    Chord { chord: Chord, bass: PitchClass },
    // ...
    /// Three or more pitch classes that are not a chord we know, named
    /// by their set class, such as "4-23" for a stack of fourths.
    Unknown(PitchSet),
}

```

One more variant, `Polychord`, is two chords at once, such as the Eb7 over Fb major that opens the Augurs of Spring in The Rite of Spring. Only the chord hearing finds those, from how the notes are grouped ([Hearing harmony: chords and keys from notes](#)); the notes alone cannot say which belong to which chord.

`analyze` takes any notes and returns a `Harmony`:

```

pub fn analyze(notes: impl IntoIterator<Item = Note>) -> Harmony {
    let mut pitch_set = PitchSet::EMPTY;
    let mut lowest: Option<Note> = None;
    for note in notes {
        pitch_set.insert(note.pitch_class());
        lowest = Some(match lowest {
            Some(low) => low.min(note),
            None => note,
        });
    }

    let Some(lowest) = lowest else {
        return Harmony::Silence;
    };
    let bass = lowest.pitch_class();

    match pitch_set.len() {
        1 => Harmony::Single(bass),
        2 => {
            let high = pitch_set.iter().find(|&p| p !=
bass).unwrap_or(bass);
            Harmony::Interval {
                low: bass,
                high,
                semitones: bass.semitones_up_to(high),
            }
        }
        _ => match find_chord(pitch_set, bass) {
            Some(chord) => Harmony::Chord { chord, bass },
            None => Harmony::Unknown(pitch_set),
        },
    }
}

```

`impl IntoIterator<Item = Note>` in an *argument* means “anything that can be looped over and gives notes”: a `Vec<Note>`, an array, or an iterator straight from a pattern ([sounding_at](#)), with nothing collected in between. One loop builds the set and finds the lowest note, and a `match` on the number of pitch classes picks the kind of answer.

Finding the chord

Any note of a chord could be its root. `find_chord` tries each one, starting with the bass, because chords are most often played with the root lowest:

```

fn find_chord(pitch_set: PitchSet, bass: PitchClass) -> Option<Chord> {
    let roots = || {
        (0..12)
            .map(move |semitones| bass.transpose(semitones))
            .filter(move |&root| pitch_set.contains(root))
    };

    let exact = roots().find_map(|root| {
        ChordQuality::identify(pitch_set.relative_to(root)).map(|quality|
Chord::new(root, quality))
    });

    exact.or_else(|| {
        roots().find_map(|root| {
            let mut with_fifth = pitch_set;
            with_fifth.insert(root.transpose(7));
            if with_fifth == pitch_set {
                return None; // the fifth was already there
            }
            ChordQuality::identify(with_fifth.relative_to(root))
                .filter(|quality| quality.intervals().len() >= 4)
                .map(|quality| Chord::new(root, quality))
        })
    })
}

```

- `roots` is a **closure that makes an iterator**. An iterator is used up as it goes, so to search twice the code calls `roots()` again for a fresh one.
- `find_map` is `find` and `map` in one: it calls the closure on each item and returns the first `Some`.
- `or_else` on an `Option` runs its closure only if the option is `None`. The second search, which allows the fifth to be missing, runs only when the exact search found nothing. Players often leave the fifth out of seventh chords, and C, E and B $\flat$ is still heard as C7.
- `Option::filter` keeps a `Some` only if the closure agrees. It stops “C and E, plus a guessed G” from being named a whole C major chord: guessing is allowed only for chords of four notes or more.

Starting with the bass settles real ambiguities. C E G A is both C6 and Am7: over a C it is C6, over an A it is Am7. C D G is both Csus2 and Gsus4. Big chords overlap all the time: Am11 and Cmaj13 are the same six pitch classes, A B C D E G, and the bass names them.

```

// C E G A is both C6 and Am7: the lowest note decides.
assert_eq!(label(&[60, 64, 67, 69]).unwrap(), "C6");
assert_eq!(label(&[57, 60, 64, 67]).unwrap(), "Am7");

```

Labels: match guards and nested patterns

The chord lane above the piano roll shows a short label for each step:

```
pub fn label(&self, spelling: Spelling) -> Option<String> {
    match *self {
        Harmony::Silence => None,
        Harmony::Single(pitch_class) =>
            Some(pitch_class.name(spelling).to_string()),
        // A root and its fifth is a "power chord", written C5.
        Harmony::Interval {
            low, semitones: 7, ..
        } => Some(format!("{}", 5", low.name(spelling))),
        // A dash, not a plus: "C+" would read as C augmented.
        Harmony::Interval { low, high, .. } => {
            Some(format!("{}", low.name(spelling),
high.name(spelling)))
        }
        Harmony::Chord { chord, bass } if bass != chord.root => {
            Some(format!("{}", chord.name(spelling),
bass.name(spelling)))
        }
        Harmony::Chord { chord, .. } => Some(chord.name(spelling)),
        // ...
        Harmony::Unknown(set) => Some(set.set_class().name()),
    }
}
```

- `semitones: 7` inside a pattern matches only that value. Other intervals fall through to the next arm.
- `..` means “and the other fields, whatever they are”.
- `if bass != chord.root` is a **match guard**: inversions get a slash, root-position chords do not.
- Arms are tried from the top, so the more specific arm comes first. If the plain `Harmony::Chord { chord, .. }` came first, the slash arm could never match.
- `match *self` matches on the value behind the reference. That works because every field is `Copy`, so the patterns copy them out.

A group with no chord name gets the name of its **set class**, such as 3-1 for three neighbouring semitones ([Pitch-class sets, and one harmony with four names](#)).

`label` names black keys with plain sharps or flats. `label_in(scale)` spells the same answer in a key, with `Scale::spell` and `Scale::spell_root`: E $\flat$ rather than D $\sharp$ in C major, A/C $\sharp$ in D minor, E $\sharp$ dim in F $\sharp$ major.

`describe(scale)` writes a sentence for the piano roll’s line about the step under the pointer, and for the status bar while you hold keys, such as “Am: A minor, chord vi of C major.” or “C/E: C major with E in the bass, chord I⁶ of C major.” It says whether a

chord is one of the key's, **borrowed** from the parallel key, or a **secondary dominant**; those words are explained in [Roman numerals, the harmony keys and chord functions](#).

Naming an interval

Two notes are an **interval**, and an interval has two names. Its size in semitones gives the everyday name, a table of twelve (`interval_name` : 3 is a “minor third”, 6 a “tritone”). But musicians name it from its spelling: the letters give the number, the semitones the quality. C to E is a third (C, D, E), and so are C to E $\flat$ and C $\sharp$ to E; a third of four semitones is major, of three minor. So C up to D $\sharp$ is an augmented second, even though it sounds like the minor third C up to E $\flat$.

```
let letters = usize::from(low.letter.steps_up_to(high.letter));
let semitones = low.pitch_class().semitones_up_to(high.pitch_class()) as i8;
// How far it is from the major or perfect interval, from -6 to 5.
let off = (semitones - MAJOR[letters] + 18).rem_euclid(12) - 6;
let perfect = matches!(letters, 0 | 3 | 4);
let quality = match (perfect, off) {
    (true, 0) => "perfect",
    (false, 0) => "major",
    (false, -1) => "minor",
    (_, 1) => "augmented",
    (true, -1) | (false, -2) => "diminished",
    (_, 2) => "doubly augmented",
    _ => "doubly diminished",
};
```

Unisons, fourths and fifths are “perfect” and have no major or minor; one semitone smaller than perfect is diminished, where for the others diminished is one smaller than minor. Adding 18 and taking `rem_euclid(12)` before subtracting 6 folds any difference into -6 to 5, so a seventh that wraps past the octave still compares correctly.

In a key, the two notes take the key's names, so in A harmonic minor F up to G $\sharp$ really is an augmented second, the scale's famous gap. A note from outside the key that would make an odd interval is named the usual way for the interval's size instead: in C major, C and E $\flat$ make a minor third.

Inversions

A chord symbol writes an inversion with a slash. A Roman numeral writes it with the figures of **figured bass**, the shorthand of Baroque keyboard players, which are the intervals above the bass:

Chord	Lowest note	Chord symbol	Numeral in C major
C major	C, the root	C	I
C major	E, the third	C/E	I ⁶ (first inversion)
C major	G, the fifth	C/G	I ⁶ ₄ (second inversion)
G7	G, the root	G7	V7
G7	B, the third	G7/B	V ⁶ ₅ (first inversion)
G7	D, the fifth	G7/D	V ⁴ ₃ (second inversion)
G7	F, the seventh	G7/F	V ⁴ ₂ (third inversion)

C/E has C a sixth above E and G a third above it, “6/3”, shortened to ⁶. A seventh chord has three intervals above its bass; the figures keep the two that tell its inversions apart, and stand in for the 7: G7/B is V⁶₅, not V7⁶₅.

`Chord::tone_of` finds which note of the chord a pitch is, walking the intervals and their tone numbers together:

```
pub fn tone_of(self, pitch_class: PitchClass) -> Option<u8> {
    let semitones = self.root.semitones_up_to(pitch_class);
    self.quality
        .intervals()
        .iter()
        .zip(self.quality.tones())
        .find(|(interval, _)| **interval % 12 == semitones)
        .map(|(_, &tone)| tone)
}
```

and `inversion_figures` turns that into figures with one `match` on a tuple:

```
match (self.tone_of(bass), triad, seventh) {
    (Some(3), true, _) => "6",
    (Some(5), true, _) => "6_4",
    (Some(3), _, true) => "6_5",
    (Some(5), _, true) => "4_3",
    (Some(7), _, true) => "4_2",
    _ => "",
}
```

Sus, sixth and ninth chords have no figures; they keep the slash name. How the figures go onto a numeral, and what an inversion changes about where a chord goes next (the cadential six-four), are in [Roman numerals, the harmony keys and chord functions](#).

One track, or all of them

The piano roll's chord lane has two rows. The first names the chords of the track being edited, step by step, with `analyze`. The band's piano may play G7 as B D F G, so on its own it is G7/B. The second row, **All**, names what the whole band plays there, beat by beat: if the bass plays F under that G7, the band plays G7/F, V⁴₂. Naming the whole band is harder than naming one track, because the melody and the bass line have passing notes that are not in the chord, and an arpeggio plays its chord one note at a time. That is the job of the chord hearing, in [Hearing harmony: chords and keys from notes](#).

Testing it

The tests read like a harmony textbook. Every quality's notes are different, every quality has a tone number for each interval, and the chords are spelled by their letters:

```
assert_eq!(
    spelled(name(Letter::C, 0), ChordQuality::Diminished7),
    "C Eb Gb Bbb"
);
assert_eq!(
    spelled(name(Letter::E, 0), ChordQuality::Dominant7Sharp9),
    "E G# B D F##"
);
```

The analysis names real voicings, in any order and octave, and without their fifth:

```
assert_eq!(label(&[48, 64, 67, 69, 74]).unwrap(), "C6/9");
assert_eq!(label(&[55, 59, 62, 65, 68]).unwrap(), "G7b9");
// E, G#, B, D and G: the "Hendrix chord".
assert_eq!(label(&[52, 56, 59, 62, 67]).unwrap(), "E7#9");
```

```
assert_eq!(label(&[48, 64, 70]).unwrap(), "C7");
```

The descriptions use the key, inversions included:

```
assert_eq!(describe(&[57, 60, 64]), "Am: A minor, chord vi of C major.");
```

And the chord symbols are read in every way of writing them, from “C” and “Am7” to “C7(b9)”, “Bø” and “Cm(maj7)”, and moved to all twelve keys by their letters.

Exercises on chords are in [Exercises: music theory in code](#).

Roman numerals, the harmony keys and chord functions

Files: `crates/lineup-core/src/harmony/scale.rs`, `crates/lineup-core/src/harmony/keys.rs`, `crates/lineup-core/src/harmony/progression.rs`, `crates/lineup-core/src/library/numeral.rs`, `crates/lineup-core/src/chord_sheet/function.rs` and `crates/lineup-app/src/harmony_keys.rs`

A chord's name says what it is; its **Roman numeral** says what it is *in the key*. The numerals are how Ferrous Lineup teaches harmony: they label the harmony keys, they sit under every chord of the chord sheet, they explain the hints about what could come next, and the library writes its progressions with them. This chapter is about writing and reading numerals, the harmony keys, the tables of common moves, and how the chord sheet works out what a chord from outside the key is doing there.

The music, briefly

A key has seven chords, one on each note of its scale, and musicians number them with Roman numerals: upper case for a major chord, lower case for a minor one, ° for diminished, + for augmented, ø for half-diminished.

I	ii	iii	IV	V	vi	vii°
C	Dm	Em	F	G	Am	Bdim

Numerals let you talk about a progression in every key at once: I vi IV V is C Am F G in C major and G Em C D in G major.

Each chord does one of three jobs in its key, its **function**:

Function	Chords	Feels like
Tonic	I, iii, vi	home: the music can rest here
Subdominant	ii, IV	leaving home, on the way to the dominant
Dominant	V, vii°	tension that pulls back home to I

The dominant's pull comes from its third, B in G major, a semitone below C: the **leading tone**, which wants to rise to the tonic. In a minor key the scale has no leading tone (in A minor, G is a whole tone below A), so music in minor keys raises it to G#, which is what the harmonic minor scale is, and its V is major: E, not Em.

A few moves between the chords are the backbone of most music:

Move	In C	Why it works
V to I	G to C	the strongest ending: the leading tone rises to the tonic
IV to V	F to G	a step up, from moving away to wanting home
ii to V	Dm to G	the root falls a fifth; ii V I is everywhere, from jazz to pop
IV to I	F to C	the soft “amen” (plagal) cadence
V to vi	G to Am	the deceptive cadence: you expect C and get its relative minor

A root that **falls a fifth** (G to C, D to G, A to D) is the strongest move there is, and many of the suggestions are of that kind.

Songs also use chords from outside the key, and harmony names what those are doing. A **borrowed** chord comes from the minor key on the same tonic, as Fm does in C major. A **secondary dominant** is the dominant of another chord of the key: D7 is the V of G, written V7/V, and for a moment it makes G sound like home. The **Neapolitan** is a major chord on the lowered second degree, D $\flat$ in C. A **pivot chord** belongs to two keys, and the music changes key through it. The manual’s [What chords do: numerals and functions](#) and [Chords from outside the key](#) teach all of this with examples to play.

Writing a numeral

A numeral has three parts: a sharp or flat in front, the numeral, and what follows it, the “7” of V7 or the “°” of vii°. All three come from one function:

```

fn numeral_on(self, letters: u8, chord: Chord) -> Option<String> {
    const NUMERALS: [&str; 7] = ["I", "II", "III", "IV", "V", "VI",
    "VII"];
    if !self.is_heptatonic() {
        return None;
    }
    let letters = usize::from(letters);
    let unaltered =
        self.unaltered_degree(letters, chord.quality,
MinorNumerals::current()) as i8;
    let semitones = self.root.semitones_up_to(chord.root) as i8;
    // How far the root is from the note the numeral names without an
    // accidental, from -6 to 5.
    let shift = (semitones - unaltered + 18).rem_euclid(12) - 6;
    let accidentals = if shift < 0 { "b" } else { "#"
}.repeat(shift.unsigned_abs() as usize);
    let numeral = if chord.quality.is_minor() {
        NUMERALS[letters].to_lowercase()
    } else {
        NUMERALS[letters].to_string()
    };
    Some(format!(
        "{accidentals}{numeral}{}",
        chord.quality.numeral_suffix()
    ))
}

```

- `letters` is the degree, how many letters above the tonic the root is. It picks the numeral: 4 letters up is V.
- `unaltered` is the note the plain numeral names, in semitones above the tonic. In a major key, the note of the major scale: V names the note seven semitones up. (In a minor key it can be another, as the next section shows.)
- `shift` is how far the chord's root is from that note. The difference of two pitch classes is only known up to octaves, so adding 18, taking the remainder by 12 and taking 6 off gives the nearer way round, from -6 to 5: a root one semitone below gives -1, a flat, not +11.
- The case comes from the quality, and the end from `numeral_suffix` ([Chords: building, naming and inversions](#)).

So in C major, B $\flat$ major is $\flat$ VII: the plain VII names B, eleven semitones above C; B $\flat$ is ten, one below, so one flat.

Which degree a chord stands on can be counted in two ways, and both are used. `roman_numeral(chord)` counts from the sound: the root's letter in the key's own spelling ([Scales, modes and keys](#)). `spelled_numeral(root, chord)` counts from the root as it was **written**, so a chord typed on the chord sheet keeps its letter. In C major, C $\sharp$ °7, a raised tonic leading up to Dm, is $\sharp$ °7, and D $\flat$ 7 is $\flat$ II7. From the sound alone both would stand on the second degree.

```
pub fn spelled_numeral(self, root: NoteName, chord: Chord) ->
Option<String> {
    self.numeral_on(self.tonic_name().letter.steps_up_to(root.letter),
chord)
}
```

Figures

An inversion's figures ([Chords: building, naming and inversions](#)) go after the numeral, in place of a seventh chord's 7: V7 with its third in the bass is V^6_5 . A secondary dominant keeps its target after the figures: V^6_5/vi .

```
pub fn with_figures(numeral: &str, figures: &str) -> String {
    if figures.is_empty() {
        return numeral.to_string();
    }
    // A secondary dominant's figures go on its own numeral, before the
    // slash:  $V^6_5/vi$ .
    let (own, target) = match numeral.split_once('/') {
        Some((own, target)) => (own, format!("{target}")),
        None => (numeral, String::new()),
    };
    let own = own.strip_suffix('7').unwrap_or(own);
    format!("{own}{figures}{target}")
}
```

`split_once('/')` splits at the first slash; `strip_suffix('7')` gives the text without a final 7, or `None` if it has none.

Reading a numeral back

The library stores its progressions as numerals, and the chord sheet accepts them typed. `numeral::read` takes a numeral apart:


```

let mut shift: i32 = 0;
let mut rest = text;
loop {
    if let Some(after) = rest.strip_prefix('b') {
        shift -= 1;
        rest = after;
    } else if let Some(after) = rest.strip_prefix('#') {
        shift += 1;
        rest = after;
    } else {
        break;
    }
}
// The numeral, in either case. `get(..len)` gives `None` instead of
// panicking when `rest` is shorter than the numeral.
let (numeral, degree) = NUMERALS.iter().find(|(numeral, _)| {
    rest.get(..numeral.len())
        .is_some_and(|start| start.eq_ignore_ascii_case(numeral))
})?;

```

`strip_prefix` gives the rest of the text when it starts with the prefix. Slicing a string with `&rest[..n]` panics when `n` is past the end; `get(..n)` gives `None` instead, so a short text is simply not that numeral. The table lists III before II and I, and VII before VI and V, so a longer numeral is tried before a shorter one it starts with. A numeral in mixed case, such as “Iv”, is refused, since its case is what says major or minor.

`parse_in(text, key, numerals)` then turns degree and shift back into semitones above the tonic, with the same `unaltered_degree` that writing uses, so writing a numeral and reading it back can never disagree. A test checks exactly that for 10,368 numerals: both ways of counting, four kinds of key, twelve tonics, a chord on each of the twelve notes and nine qualities.

Minor keys: bIII or III

In A minor, the chord C major has two names. Pop and jazz books call it **bIII**; classical harmony books call it **III**. Both are right, and **Edit > Settings... > Harmony > Roman numerals in minor keys** lets you choose.

Counted from the major scale, the numerals are measured against the major scale on the same tonic, A major, whose third is C♯. C is a semitone below, so its chord is **bIII**; F is **bVI** and G **bVII**. One table of progressions then works in every key and mode: **bVI bVII i** is the same progression in A minor and A Dorian, and a song in C major that borrows A♭ and B♭ from C minor writes them **bVI** and **bVII** too. The library always writes its progressions this way.

Counted from the minor scale, the numerals are measured against the key's own scale. C is simply its third note, so its chord is III.

Only the third, sixth and seventh degrees differ, the notes that differ between A major and A minor. But the sixth and seventh notes of a minor key also move: harmonic minor raises G to G $\sharp$, melodic minor raises F to F $\sharp$ as well. So a plain numeral on those degrees must name one of two notes, and the rule classical books use is: **a plain numeral names the form whose chord belongs to the key**. VII is G major, from natural minor; vii $^\circ$ is G $\sharp^\circ$, the leading-tone chord of harmonic minor, since G $^\circ$ (G B $\flat$ D $\flat$) is no chord of A minor.

	from the major scale						from the minor scale							
natural minor	i	ii°	bIII	iv	v	bVI	bVII	i	ii°	III	iv	v	VI	VII
harmonic minor	i	ii°	bIII+	iv	V	bVI	vii°	i	ii°	III+	iv	V	VI	vii°
melodic minor	i	ii	bIII+	IV	V	vi°	vii°	i	ii	III+	IV	V	vi°	vii°

That rule is `unaltered_degree` :

```

pub(crate) fn unaltered_degree(
    self,
    letters: usize,
    quality: ChordQuality,
    numerals: MinorNumerals,
) -> u8 {
    use ScaleKind::{HarmonicMinor, MelodicMinor, NaturalMinor};
    if numerals == MinorNumerals::FromMajor || !self.is_minor_key() {
        return ScaleKind::Major.tonal_intervals()[letters];
    }
    let natural = NaturalMinor.tonal_intervals()[letters];
    if letters < 5 {
        return natural;
    }
    // Whether the chord on `semitones` is in one of these scales,
    // counted from C, since only the distance from the tonic counts.
    let of_key = |semitones: u8, kinds: &[ScaleKind]| {
        let chord =
            Chord::new(PitchClass::C.transpose(i32::from(semitones)), quality);
        kinds
            .iter()
            .any(|&kind| Scale::new(PitchClass::C,
kind).is_diatonic(chord))
    };
    if !of_key(natural, &[NaturalMinor]) && of_key(natural + 1, &
[HarmonicMinor, MelodicMinor])
    {
        natural + 1
    } else {
        natural
    }
}

```

For the first five degrees the natural minor note is the answer. For the sixth and seventh, the function asks the rule's question: is the chord on the natural note a chord of natural minor? If not, and the chord a semitone higher is in harmonic or melodic minor, the raised note is the one meant. `of_key` builds the chord on C, because only its distance from the tonic matters.

The setting changes how numerals are **written** and **read**, never a chord. A song stores chords, not numerals, so it is the same whichever way you choose. Major keys and the modes are written the same both ways.

A setting every function can see, one per thread

Numerals are written in many places, the harmony keys, the chord sheet, the piano roll, the hints, a video export, and all of them end up in `numeral_on`. Passing the setting down as a parameter would change the signature of dozens of functions on

the way, for a choice that changes once in a while. So the setting is kept where `numeral_on` can read it:

```
thread_local! {
    /// How the numerals written on this thread count a minor key's
    /// degrees. The window's thread sets it from the settings each frame;
    /// each thread has its own, so tests running side by side never see
    /// each other's.
    static MINOR_NUMERALS: Cell<MinorNumerals> = const {
        Cell::new(MinorNumerals::FromMajor) };
}

impl MinorNumerals {
    /// How numerals are written on this thread.
    pub fn current() -> MinorNumerals {
        MINOR_NUMERALS.with(Cell::get)
    }

    /// Writes numerals this way on this thread from now on.
    pub fn set(self) {
        MINOR_NUMERALS.with(|numerals| numerals.set(self));
    }
}
```

The window's thread sets it before drawing each frame:

```
// Every view writes numerals as the settings say.
settings::minor_numerals(ctx).set();
```

Two pieces of Rust make this safe:

- `thread_local!` gives each thread its own copy of the value. `cargo test` runs tests on many threads at once, and a test that writes numerals from the minor scale must not change what a test running beside it sees. With an ordinary `static` it would, and the other test would fail now and then, depending on timing.
- `Cell` lets a value be changed through a shared reference. A thread-local value is only ever reached through `&`, so a plain value could never be set. `Cell` works by moving whole values in and out (`get`, `set`), never handing out a reference to what is inside, so nothing can be read while it changes.

Because the value belongs to a thread, a new thread starts with the default. A video export draws its views on a thread of its own, so it takes the setting along, with the language and the note naming ([Exporting: WAV, MIDI and video](#)):

```
let numerals = MinorNumerals::current();
```

and calls `numerals.set()` first thing on the new thread.

Putting it back, even after a panic

Tests write numerals one way for a while, then go back:

```
pub fn with<T>(self, f: impl FnOnce() -> T) -> T {
    struct Restore(MinorNumerals);
    impl Drop for Restore {
        fn drop(&mut self) {
            self.0.set();
        }
    }
    let _restore = Restore(MinorNumerals::current());
    self.set();
    f()
}
```

`Restore` holds the old value, and its `Drop` puts it back. Rust drops a local variable when it goes out of scope, and also while a **panic** unwinds through the function. A failing `assert!` inside `f` is a panic, so even then the old setting comes back, and the next test on that thread starts clean. Cleaning up in a `Drop` like this is called a **guard**; it is how the standard library's `MutexGuard` unlocks a mutex.

The variable is called `_restore`, not `_`. A plain `_` does not bind at all, so the guard would be dropped at once, on the same line, before `self.set()`, and nothing would put the old value back after `f`.

The hints are sentences written once, with numerals as the library writes them: “I to IV: moving away from home”. When numerals are counted from the minor scale, `numeral::written_in` rewrites the numerals inside such a sentence. Most of the time nothing changes, so it returns a `Cow<str>`: the original text borrowed, or a new `String` only when it had to rewrite something.

The harmony keys

The harmony keys along the bottom of the window are one big key for each chord of the key, coloured by function, played by holding them down. The model is pure theory, so it lives in `lineup-core`, with tests, and knows nothing of egui:

```

pub fn chord_keys(scale: Scale, size: ChordSize) -> Vec<ChordKey> {
    let scale = chord_scale(scale);
    let leading_tone = Scale::new(scale.root, ScaleKind::HarmonicMinor);
    (0..7)
        .filter_map(|degree| {
            let from = if scale.kind == ScaleKind::NaturalMinor && degree
== 4 {
                leading_tone
            } else {
                scale
            };
            let chord = chord_on(from, degree, size)?;
            Some(ChordKey {
                degree,
                chord,
                numeral: scale.roman_numeral(chord)?,
                function: has_functions(scale.kind).then(||
Function::of(degree)),
            })
        })
        .collect()
}

```

- `filter_map` runs the closure on every degree and keeps the `Some` s, and inside the closure `?` turns a `None` into “leave this one out”.
- In a natural minor key, the chord on V comes from harmonic minor: major, with the leading tone.
- `chord_scale` gives a pentatonic or blues scale, which has no seven chords of its own, the major or minor scale on the same root.
- `bool::then` turns `true` into `Some` of the closure’s value and `false` into `None` . The modes get no function: they have no dominant that pulls home the way V does. In C Locrian the chord on the fifth note is G $\flat$ major, which does not pull to C at all.

`Function::of` is the table of jobs by degree:

```

pub fn of(degree: usize) -> Function {
    match degree % 7 {
        1 | 3 => Function::Subdominant,
        4 | 6 => Function::Dominant,
        _ => Function::Tonic,
    }
}

```

`ChordSize` picks triads, sevenths or ninths. On two degrees of a major key, iii and vii, the ninth lands a semitone above the root, an octave up; that clashes, and players leave it out, so `Scale::ninth` gives `None` there and the key falls back to the seventh chord:

```
ChordSize::Ninths => scale.ninth(degree).or_else(||
scale.seventh(degree)),
```

`Option::or_else` tries the closure only when the first value is `None`: “if that did not work, try this”.

Playing a key

Pressing C and then F could play C E G and then F A C, every note jumping up. A pianist would keep the C and move the other two fingers a step each, to C F A.

`keys::play` voices each chord close to the one before (the voicing itself is the band’s, in [How the band plays a chord sheet](#)) and adds the root below, as a left hand would:

```
pub fn play(chord: Chord, previous: Option<&[Note]>) -> (Vec<Note>,
Vec<Note>) {
    let voicing = voice(chord, previous);
    let mut notes = voicing.clone();
    if let Some(&lowest) = voicing.first() {
        // The root below the lowest note: an octave below if the lowest
        // note is the root itself.
        let down = match chord.root.semitones_up_to(lowest.pitch_class())
{
            0 => 12,
            gap => gap,
        };
        if let Some(root) = lowest.transpose(-i32::from(down)) {
            notes.insert(0, root);
        }
    }
    (voicing, notes)
}
```

It returns a tuple of two vectors: the voicing, to remember for next time, and the notes to play. The caller takes it apart with `let (voicing, notes) = ...`.

A harmony key is a key, not a button: its chord starts when the mouse goes down and stops when it comes up, and the number keys 1 to 7 play them too. Which one holds a key down is an enum whose variant carries the number key:

```
enum Holder {
    Mouse,
    /// A number key on the computer keyboard, 1 to 7.
    Digit(Key),
}

/// The harmony key that is down, and the notes it started.
#[derive(Debug, Clone, PartialEq)]
struct Down {
    key: usize,
    holder: Holder,
    notes: Vec<Note>,
}
```

Only the holder that started a chord can stop it, and `Down` keeps the notes it started, so the right notes stop even if the key or the chord size changed while it was held. The notes go the same way as the typing keyboard's ([MIDI: messages, devices, recording and files](#)), so they sound on the selected channel, are named in the status bar and are recorded, without that code knowing the harmony keys exist.

Which key lights up

While the music plays, the key of the chord you hear is lit and framed in white, the most visible mark on the bar. Keys held down win; else the chord is the one heard over the beat at the playhead, from the chord hearing ([Hearing harmony: chords and keys from notes](#)), so an arpeggio lights its chord and two chords at once light two keys. A chord lights a key when it has the same root and the same kind of triad as the key's chord there:

```
pub fn key_for(keys: &[ChordKey], chord: Chord) -> Option<usize> {
    keys.iter().position(|key| {
        key.chord.root == chord.root && key.chord.quality.family() ==
        chord.quality.family()
    })
}
```

G7 lights V in C major. D7 lights nothing: the key's chord on D is minor, and D7 comes from outside the key.

What comes next: knowledge as data

After a chord, the keys that often come next get a thin yellow outline, the most usual one a little brighter, and a hint line says why. `next_chords(scale, current)` gives

those chords, most usual first. The knowledge is not in code but in tables: each chord is found by its root (semitones above the tonic) and its kind of triad, and lists the moves that often follow it.

```
struct Move {
    to: u8,
    short: &'static str,
    reason: &'static str,
}

const fn go(to: u8, short: &'static str, reason: &'static str) -> Move {
    Move { to, short, reason }
}

struct Slot {
    root: u8,
    family: ChordFamily,
    moves: &'static [Move],
}
```

```
const MAJOR: &[Slot] = &[
    slot(
        0,
        Major,
        &[
            go(
                5,
                "away from home",
                "I to IV: moving away from home, toward the dominant.",
            ),
            go(
                7,
                "tension, wants to go home",
                "I to V: tension that wants to come back to I.",
            ),
            // ...
        ],
    ),
    // ...
];
```

A constant can only be built by code the compiler can run while compiling, and a `const fn` is such code, so the tiny `go` and `slot` keep each move on a line or two. `MAJOR` is a slice of slots, so it holds as many entries as a key needs, the borrowed iv, bIII, bVI and bVII included. All of it is built into the program. Changing what the app suggests means editing a table, and the tests check the table: every degree of three kinds of key has suggestions, and no chord suggests itself.

Each move carries its reason twice, in a sentence and in a few words. The hint line's level, **Detailed**, **Short** or **Minimal**, picks one:

```
pub fn why(&self, level: HintLevel) -> Option<&str> {
    match level {
        HintLevel::Detailed => Some(&self.reason),
        HintLevel::Short => Some(&self.short),
        HintLevel::Minimal => None,
    }
}
```

A `Suggestion` keeps its reasons as `Cow<'static, str>`, "clone on write": usually a borrowed `&'static str` from the table, with no allocation, and an owned `String` only when `written_in` had to rewrite its numerals.

Minor keys find their chords in another table, natural minor with the major V and the leading-tone vii° of harmonic minor:

```
const MINOR_CHORDS: [(u8, ChordQuality); 8] = [
    (0, ChordQuality::Minor),
    (2, ChordQuality::Diminished),
    (3, ChordQuality::Major),
    (5, ChordQuality::Minor),
    (7, ChordQuality::Major),
    (8, ChordQuality::Major),
    (10, ChordQuality::Major),
    (11, ChordQuality::Diminished),
];
```

Secondary dominants first

A chord from outside the key that sounds like a dominant is most often the dominant of the chord a fifth below it, so `next_chords` checks for that before looking at the tables:

```

let dominant_kind = current.quality == ChordQuality::Major ||
current.quality.is_dominant();
let target = (root + 5) % 12;
if dominant_kind
  && !scale.is_chord_of_key(current)
  && target != 0
  && let Some(suggestion) = suggest(target, SECONDARY_DOMINANT_SHORT,
SECONDARY_DOMINANT)
  && suggestion.chord.quality != ChordQuality::Diminished
{
  return vec![suggestion];
}

```

Read the let chain as the rule it is: the chord sounds like a dominant, it is not one of the key's own chords, the chord a fifth below is not the tonic (in a minor key, a major chord on V is the key's own dominant), there is a chord of the key there, and it is not diminished, which is too unstable to sound like home even for a moment. A fifth down and a fourth up are the same pitch class, so the target is `(root + 5) % 12`. In C major D major suggests G, A7 Dm, E Am, and C7, whose B $\flat$ makes it no longer the tonic chord, F.

The bass matters: the cadential six-four

C/G, the tonic chord with its fifth in the bass, sounds unsettled: over G, the C and the E want to fall a step to B and D, which makes G. So in a cadence C/G goes on to V, and is called the **cadential six-four** (I^6_4). `next_chords_over` takes the bass too, and handles that case before handing everything else to `next_chords`:

```

let six_four = current == Chord::new(scale.root, tonic) &&
current.tone_of(bass) == Some(5);
if !six_four {
  return next_chords(scale, current);
}

```

What a chord does: the chord sheet's functions

The chord sheet writes a numeral under every chord, and for a chord from outside the key, what it is doing there. That depends on more than the chord and the key: a secondary chord is named by where it goes, and a pivot by the chords around a key change. So `Functions::read` reads the whole sheet in one pass, and the sheet asks it about each chord. What a chord does is a `Role`, an enum whose variants carry what they need:

```
pub enum Role {
  /// One of the key's own chords, or a chord outside it that none of
  /// the others names.
  Plain,
  /// Taken from the parallel key, `from`.
  Borrowed { from: Scale },
  /// The dominant of `of`, one of the key's chords.
  SecondaryDominant { of: Chord },
  /// The chord on the leading tone of `of`, one of the key's chords.
  SecondaryLeadingTone { of: Chord },
  /// A major chord on the lowered second degree.
  Neapolitan,
}
```

A `match` gets the data out where it is used, so `of` only exists in the arms where it means something:

```
match self.role {
  Role::SecondaryDominant { of } => Some(with_figures(
    &format!(
      "V{}/{}",
      chord.quality.numeral_suffix(),
      self.target_numeral(of)?
    ),
    figures,
  )),
  // ...
  Role::Neapolitan => Some(with_figures("N", figures)),
  Role::Plain | Role::Borrowed { .. } => figured(self.key, self.symbol),
}
```

`Role::Borrowed { .. }` matches a borrowed chord whatever key it comes from.

The key's helpers

The questions behind the roles are methods of `Scale`:

- `parallel()`: the key with the same tonic and the other mode, C minor for C major.
- `is_chord_of_key(chord)`: in the scale, or in a minor key the V or vii° with the raised seventh of harmonic minor. E7 and G#dim7 belong to A minor.
- `borrowed_from(chord)`: the parallel key, if the chord is not the key's own but is that key's. A major key borrows from harmonic minor too: Bb and Bdim7 in C major both come from C minor.
- `secondary_dominant_target(chord)`: for a major or dominant seventh chord from outside the key, the key's major or minor chord (not the tonic) a fifth below

it: E7 in C major is the V7 of Am.

- `secondary_leading_tone_target(chord)` : for a diminished chord, the chord a semitone above it: F#°7 in C major leads to G.

Which role comes first

Some chords could be two things. In A minor, A major is the dominant of Dm and also the major tonic chord borrowed from A major; D major is the IV of melodic minor and the dominant of G. `role` decides by where the chord goes:

```
let goes_to = |of: Chord| next.is_some_and(|next| arrives(next, of));
let dominant = key.secondary_dominant_target(chord);
let leading = key.secondary_leading_tone_target(chord);
let secondary = match (dominant, leading) {
  (Some(of), _) => Some(Role::SecondaryDominant { of }),
  // A leading-tone chord is named after where it goes: F#°7 going
  // on to B7 in C major is the ii of E minor, not the vii of G.
  (_, Some(of)) if goes_to(of) => Some(Role::SecondaryLeadingTone { of
}),
  _ => None,
};
// A chord that is both, such as A major in A minor (the major tonic,
// or the V of Dm), is a secondary dominant where it goes to the
// chord it leads to, and a borrowed chord anywhere else.
if let Some(of) = dominant.or(leading)
  && goes_to(of)
{
  return secondary.unwrap_or(Role::Plain);
}
// So is a chord of melodic minor, such as D major in A minor: the V
// of G going to G, and the key's own IV anywhere else.
if in_minor(key, chord, &[MelodicMinor]) {
  return Role::Plain;
}
if let Some(from) = borrowed_from(key, symbol) {
  return Role::Borrowed { from };
}
if is_neapolitan(key, symbol) {
  return Role::Neapolitan;
}
secondary.unwrap_or(Role::Plain)
```

Before this, `role` has already sent away the chords that need no name beyond their numeral: every chord in a mode (E major in E Phrygian is the tonic chord of flamenco, not the dominant of A minor), every chord of the key, and in a minor key every chord of its natural or harmonic minor. A chord “goes to” its target when the next chord is on that root, or is a six-four chord over it, which holds the target back for a moment.

`is_some_and` answers `false` for `None`, so the last chord, with nothing after it, goes nowhere.

A borrowed chord must be **spelled** as the other key writes it, not only sound like it. G augmented (G B D $\sharp$) is played on the same keys as E $\flat$ augmented (E $\flat$ G B), a chord of C harmonic minor. But its D $\sharp$ rises to E, where an E $\flat$ would fall: it is no note of C minor. So `borrowed_from` checks that every note of the chord is written on the letter the key's scales give it, counting up from the tonic. In the same way the Neapolitan must be spelled on the second letter: D $\flat$ major in C is one, C $\sharp$ major, the same keys, is a chromatic chord on the tonic.

Finding the pivot

A **modulation** is a move to a new key that the music stays in. In C major Am is vi; in G major it is ii. In C, Am, D7, G the ear hears Am as vi in C; then D7 comes, which C major does not have, and the ear hears that Am was really ii in G. Am is the **pivot chord**, and the chord sheet writes both its numerals, old key first: `vi = ii`.

`modulation` looks at the chords around each key change, with two closures that say what a chord is to the two keys:

```
let new_only = |chord: &Function| {
  let chord = chord.symbol.chord();
  belongs(to, chord) && !heard_in(from, chord)
};
let common = |chord: &Function| {
  let chord = chord.symbol.chord();
  belongs(to, chord) && heard_in(from, chord)
};
```

`heard_in` counts the old key's borrowed chords as its own, which is why Fm, borrowed in C major, can be the pivot into A $\flat$ major, where it is vi. The **arrival** is the first chord that belongs to the new key alone (D7 above). It is looked for just before the key marker first, since musicians mark a new key where they feel it has arrived, which can be a bar or two late, and then after it. The pivot is the chord before the arrival, if it belongs to both:

```
change.pivot = arrival
  .checked_sub(1)
  .map(|before| range[before])
  .filter(|before| common(before))
  .map(|before| (before.start, before.symbol));
```

`checked_sub(1)` gives `None` when the arrival is the first chord, so no index ever goes below zero, and a change that starts on its arrival has no pivot. Not every key change has one: a **direct modulation** goes straight to a chord of the new key, as a song does when it moves up a semitone for its last chorus, and a **change of mode**, C major to C minor, keeps the tonic and needs no chord to lead there.

Testing it

The tests are small chord sheets in the style of a harmony textbook, each checking the numerals and what the tooltips say:

- C Am A $\flat$ Fm G7 C: $\flat$ VI and iv are borrowed from C minor.
- C E7 Am D7/F $\sharp$ G7 C A7 Dm: V7/vi, V $^6_5$ /V and V7/ii, each going to its chord. E7 going to F is still V7/vi, and says it goes elsewhere.
- F $\sharp$ dim7, G $\sharp$ dim and C $\sharp$ m7 $\flat$ 5: vii $^\circ$ 7/V, vii $^\circ$ /vi and vii $\emptyset$ 7/ii. In Autumn Leaves read in C major, F $\sharp$ m7 $\flat$ 5 going to B7 keeps $\sharp$ iv $\emptyset$ 7.
- D $\flat$ /F in C minor is N ^{6} ; C $\sharp$ in C major is not a Neapolitan.
- In A minor, B7 is V7/V, or V7/v going to Em; A7 going to Dm is V7/iv; A major at the end is a Picardy third.
- G augmented in C major is not borrowed; Bdim7, whose A $\flat$ is C minor's, is.
- C, Am, D7, G into G major, with the marker on the Am, the D7 or the G: Am is the pivot each time.

In `scale.rs`, the minor-key numerals are checked both ways for natural, harmonic and melodic minor, and the 10,368 numerals are written and read back. In `keys.rs` and `progression.rs`, the harmony keys of every kind of key and the suggestions after every degree; for instance, in C major:

```
// D major is V of G; A7 is V of Dm; E is V of Am; C7 is V of F.
```

Exercises on numerals and functions are in [Exercises: music theory in code](#).

Editing tools as plain functions

Files: `crates/lineup-core/src/tools.rs` and `crates/lineup-app/src/piano_roll.rs`

The piano roll's **Tools** menu changes many notes at once: it moves them up a semitone, an octave or a note of the key, earlier or later, onto the grid or into the key, inverts chords and turns them into arpeggios. Each tool is a function in `lineup_core::tools` that takes some notes and returns new ones. None of them knows about windows, selections or undo, which is what makes them easy to read and to test. This chapter goes through them one by one; how the menu and the arrow keys pick a tool is in [The editors: channel rack, piano roll and playlist](#).

The music, briefly

Transposing moves music up or down without changing its shape: every interval stays the same, so a melody moved up a tone is the same melody, higher. C, E, G up a tone is D, F $\sharp$, A: still a major chord.

Moving notes **in the key** is different. Each note goes to the next note of the *scale*, not up a fixed number of semitones, so the intervals change to fit the key. In C major, C, E, G up one note of the key is D, F, A: the chord on the next degree, D minor, not D major. Composers have always used this: a short figure repeated a note higher each time, staying in the key, is a **sequence**, and Baroque music is full of them. Copy a phrase, paste it later and move it up a note in the key, and you have written one.

Quantizing moves notes onto the grid. A person never plays exactly on the beat, and a MIDI file of a performance keeps every small difference; quantizing makes the notes as exact as a drum machine.

An **inversion** of a chord has the same notes with another one in the bass, as [Chords: building, naming and inversions](#) explains. An **arpeggio** plays a chord's notes one after another, the way a guitarist picks a chord or a harp plays one, instead of all at once.

Notes in, notes out

Every tool has the same shape. It borrows the notes as a slice, `&[NoteEvent]`, and returns new ones; it changes nothing itself. The module says so at the top:


```

    //! Each tool is a plain function: it takes some notes and returns new
    //! ones, and changes nothing itself. The piano roll hands the result to
    //! [Pattern::replace_notes](crate::Pattern::replace_notes), through
    //! the app's edit, so every tool can be undone like any other change.

```

The piano roll picks the function for each tool, with a name for the undo history:

```

let (label, new) = match tool {
    // ...
    Tool::Transpose(semitones) => {
        ("transpose notes", tools::transpose(&targets, semitones))
    }
    Tool::InKey(steps) => (
        "move notes in key",
        tools::transpose_in_key(&targets, scale, steps),
    ),
    Tool::Shift(steps) => ("move notes", tools::shift(&targets, steps,
length)),
    Tool::Quantize => ("quantize notes", Some(tools::quantize(&targets,
length))),
    Tool::SnapToKey => (
        "snap notes to key",
        Some(tools::snap_to_scale(&targets, scale)),
    ),
    Tool::Invert { up } => ("invert chords", tools::invert(&targets, up)),
    // ...
};

```

`targets` is the selection, or every note of the channel when nothing is selected. The new notes then replace the old ones in one `ReplaceNotes` action, which is one step of undo; how that works, and why the selection holds notes rather than positions, is in [The editors: channel rack, piano roll and playlist](#). Keeping the tools out of the app also means the same functions serve other callers: the twelve-tone tests group a written row into chords with `tools::chords`, and the MIDI keyboard snaps keys with `tools::nearest_in_key`.

Most tools build each new note from an old one with **struct update syntax**:

```

Some(NoteEvent {
    pitch: note.pitch.transpose(semitones)?,
    ..*note
})

```

`..*note` means “every other field from `*note`”: the start, the length, the velocity, the nudge and the trim stay as they were, and only the pitch is new. `NoteEvent` is `Copy`, so `*note` copies the old note out of the slice without taking it away. If a field is added to `NoteEvent` one day, every tool keeps it without a change.

Grouping notes into chords: chords

Inverting and arpeggiating work on chords, so the first tool finds them. Here a chord is simply the notes that start on the same step:

```
pub fn chords(notes: &[NoteEvent]) -> Vec<Vec<NoteEvent>> {  
    let mut sorted = notes.to_vec();  
    sorted.sort_by_key(|note| (note.start, note.pitch));  
    // `chunk_by` cuts the sorted list wherever the start changes.  
    sorted  
        .chunk_by(|a, b| a.start == b.start)  
        .map(<[NoteEvent]>::to_vec)  
        .collect()  
}
```

Sorting by the tuple `(note.start, note.pitch)` sorts by start, and by pitch among notes that start together, so each chord comes out lowest note first. `chunk_by` then cuts the list into runs, as the chord lane does (see [The chord lane: `chunk\_by`, then `measure`](#)).

Each run is a slice that borrows from `sorted`, and `sorted` is dropped when the function returns. Returning the slices would leave them pointing at freed memory, which the borrow checker refuses, so each one is copied into a `Vec` of its own.

`<[NoteEvent]>::to_vec` is the slice method `to_vec` named as a function, so `map` can take it directly instead of `|run| run.to_vec()`. The angle brackets are needed because `[NoteEvent]` is a type, not a name: `[NoteEvent]::to_vec` would not parse.

When a tool cannot work: collecting Options

Transposing a chord near the top of the keyboard can push a note past MIDI note 127. Leaving that note where it is would break the chord; dropping it would lose a note. So the tool does neither: it refuses the whole move.

```
pub fn transpose(notes: &[NoteEvent], semitones: i32) ->
Option<Vec<NoteEvent>> {
    // Collecting an iterator of `Option`s into `Option<Vec<_>>` gives
    // `None` as soon as one item is `None`, and the whole `Vec`
    otherwise.
    notes
        .iter()
        .map(|note| {
            Some(NoteEvent {
                pitch: note.pitch.transpose(semitones)?,
                ..*note
            })
        })
        .collect()
}
```

`Note::transpose` returns `None` for a note off the keyboard (see [Pitch classes, notes and spelling](#)). The `?` inside the closure returns `None` from the *closure*, not from `transpose`, so the iterator yields an `Option<NoteEvent>` for each note. `collect` can build many kinds of collection, and the return type asks for `Option<Vec<NoteEvent>>`: that collection stops at the first `None` and is `None` itself, or holds every note. `Result` works the same way: collected into a `Result<Vec<_>, E>`, it stops at the first error.

The piano roll does nothing when a tool returns `None`, so pressing ↑ on a chord that has a note in the top row simply does not move it:

```
let Some(new) = new.filter(|new| fits_view(new)) else {
    return;
};
```

`Option::filter` turns `Some` into `None` when the test fails; here the test is that every note fits the rows the piano roll shows, which are narrower than the 128 notes of MIDI.

Moving in time: `checked_add_signed` and `then_some`

← and → move notes a step earlier or later, and with Shift four steps, a beat of 4/4. Starts are `usize`, which cannot be negative, while the move can go either way, so it is an `isize`:

```
pub fn shift(notes: &[NoteEvent], steps: isize, pattern_length: usize) ->
Option<Vec<NoteEvent>> {
    notes
        .iter()
        .map(|note| {
            let start = note.start.checked_add_signed(steps)?;
            (start < pattern_length).then_some(NoteEvent { start, ..*note
        })
    })
    .collect()
}
```

`usize::checked_add_signed` adds a signed number to an unsigned one and returns `None` if the result would fall below zero, so a note at step 0 cannot move earlier. `then_some` turns the `bool` into an `Option: Some` of the moved note if it still starts inside the pattern, `None` if it would start at its end or after. Either way the whole selection stays where it was, with the same `collect` as before.

Back onto the grid: quantize

Quantize to the steps moves each note's start and end to the nearest step. A note keeps its exact place as a step plus a `nudge` and a `trim`, in ticks, 24 to a step (see [The project: patterns, channels, tracks, clips and markers](#)), and the rounding is on the note itself:

```
/// The note moved to the nearest steps, both its start and its end:
/// what *quantizing* does. A note shorter than a step keeps one.
pub fn quantized(self) -> NoteEvent {
    let nearest = |tick: usize| (tick + TICKS_PER_STEP / 2) /
TICKS_PER_STEP;
    let start = nearest(self.start_tick());
    let end = nearest(self.end_tick()).max(start + 1);
    NoteEvent {
        velocity: self.velocity,
        ..NoteEvent::new(self.pitch, start, end - start)
    }
}
```

Integer division rounds down, so adding half a step first rounds to the nearest step.

`NoteEvent::new` makes a note on the grid, with no nudge and no trim, and the struct update takes everything from it but the velocity. The tool adds one rule: a note that would now start at the end of the pattern starts on its last step instead.

```
pub fn quantize(notes: &[NoteEvent], pattern_length: usize) ->
Vec<NoteEvent> {
    let last = pattern_length.saturating_sub(1);
    notes
        .iter()
        .map(|note| {
            let quantized = note.quantized();
            let start = quantized.start.min(last);
            NoteEvent {
                start,
                length: quantized.end() - start,
                ..quantized
            }
        })
        .collect()
}
```

Quantizing is a choice, not something the app does on its own, because it can change the music. Eighth-note triplets start every 32 ticks, a step and a third apart, and no grid of sixteenths holds them: quantized, they land on steps 0, 1 and 3 and are no longer even. That is why notes imported from a MIDI file keep their nudge until you ask. (Notes you record are put on the grid as they are recorded, by a function of their own: see [MIDI: messages, devices, recording and files](#).)

Staying in the key

Two tools work with the key: one moves notes a note of the key up or down, the other moves every note that is outside the key into it. Both use the key the song is in where the pattern plays, after key markers (see [Scales, modes and keys](#)).

One note of the key up: find_map

The next note of the key above a note is the first note, a semitone higher, then two, and so on, that is in the scale:

```
pub fn step_in_key(note: Note, scale: Scale, up: bool) -> Option<Note> {
    let direction = if up { 1 } else { -1 };
    (1..=12).find_map(|distance| {
        note.transpose(direction * distance)
            .filter(|candidate| scale.contains(candidate.pitch_class()))
    })
}
```

`find_map` calls the closure for each distance and stops at the first `Some`. The closure transposes, which can fail at the edge of the keyboard, and `filter` keeps the note only if the scale contains it. Twelve semitones reach every pitch class, so if the scale has any note at all, one is found: at worst the same note an octave away, when the scale holds nothing else. A range is lazy, and `find_map` stops at the first hit, so a step in C major tries one or two distances, not twelve.

Moving several notes of the key is that step, repeated:

```
pub fn transpose_in_key(notes: &[NoteEvent], scale: Scale, steps: i32) ->
Option<Vec<NoteEvent>> {
    notes
        .iter()
        .map(|note| {
            let mut pitch = note.pitch;
            for _ in 0..steps.unsigned_abs() {
                pitch = step_in_key(pitch, scale, steps > 0)?;
            }
            Some(NoteEvent { pitch, ..*note })
        })
        .collect()
}
```

`unsigned_abs` gives the size of `steps` as an unsigned number, so the loop runs as many times going down as going up. A note outside the key lands on the nearest note of the key in the direction of the move: C# up one step in C major is D. **Alt+↑** and **Alt+↓** use this tool.

The nearest note of the key: snapping

Snapping looks both ways at once:

```
pub fn nearest_in_key(note: Note, scale: Scale) -> Note {
    if scale.contains(note.pitch_class()) {
        return note;
    }
    (1..=6)
        .find_map(|distance| {
            [-distance, distance].into_iter().find_map(|semitones| {
                note.transpose(semitones)
                    .filter(|candidate|
                        scale.contains(candidate.pitch_class()))
            })
        })
        .unwrap_or(note)
}
```

For each distance, the inner `find_map` tries the note below before the one above, so when two notes of the key are as near, the lower one wins: in C major, C# snaps to C, F# to F and Bb to A. A rule is needed, since neither is more correct, and this one is easy to remember. Six semitones up or down reach all twelve pitch classes, so any scale with a note in it is found; only at the very ends of the keyboard can both tries fail, and `unwrap_or(note)` then leaves the note where it is. Snapping never refuses the way a move does; at worst a note stays put.

`snap_to_scale` maps it over many notes, for **Snap to the key** in the Tools menu:

```
pub fn snap_to_scale(notes: &[NoteEvent], scale: Scale) -> Vec<NoteEvent>
{
    notes
        .iter()
        .map(|note| NoteEvent {
            pitch: nearest_in_key(note.pitch, scale),
            ..*note
        })
        .collect()
}
```

The same two functions do the piano roll's **Snap to scale** button: a note you draw lands on `nearest_in_key` of the row you clicked, and a group you drag is snapped with `snap_to_scale` once it has moved (see [Moving a group](#)). A key played on the typing keyboard or a MIDI keyboard is snapped the same way (see [MIDI: messages, devices, recording and files](#)).

Inversions

Invert chords up moves each chord's lowest note above its highest; **Invert chords down** moves the highest below the lowest. Both move a note by octaves until it passes the other end of the chord:

```
/// `note` moved up by octaves until it is above `ceiling`.
fn above(note: Note, ceiling: Note) -> Option<Note> {
    let mut note = note;
    while note <= ceiling {
        note = note.transpose(12)?;
    }
    Some(note)
}
```

"Until it is above", not "one octave up": in C, E, G, C the low C must go up two octaves to be above the high C. The `?` gives up if the note would leave the keyboard.

```

pub fn invert(notes: &[NoteEvent], up: bool) -> Option<Vec<NoteEvent>> {
    let mut result = Vec::with_capacity(notes.len());
    for mut chord in chords(notes) {
        if let (Some(&lowest), Some(&highest)) = (chord.first(),
chord.last())
            && chord.len() >= 2
        {
            if up {
                chord[0].pitch = above(lowest.pitch, highest.pitch)?;
            } else {
                let last = chord.len() - 1;
                chord[last].pitch = below(highest.pitch, lowest.pitch)?;
            }
            chord.sort_by_key(|note| note.pitch);
        }
        result.extend(chord);
    }
    Some(result)
}

```

`chords` returns each chord lowest note first, so `first` and `last` are the bass and the top. They return `Option`s, because an empty `Vec` has neither, and matching a tuple of both takes them in one pattern. The `&` in `Some(&lowest)` copies the note out of the reference that `first` returns, so `lowest` and `highest` are plain `NoteEvent`s rather than borrows of `chord`. The `&&` makes a **let chain**: the body runs only if both notes exist *and* the chord has two notes or more, so a single note is left alone. `for mut chord` makes each chord a variable that can be changed, without a `let mut` inside the loop.

`inversion`, a sibling for a list of pitches, gives a chord in any inversion at once. The piano roll's chord stamp uses it for the inversion menu next to **Draw** (see [Chords: building, naming and inversions](#)):

```

pub fn inversion(pitches: &[Note], times: usize) -> Vec<Note> {
    let mut notes = pitches.to_vec();
    notes.sort();
    for _ in 0..times % notes.len().max(1) {
        let (Some(&lowest), Some(&highest)) = (notes.first(),
notes.last()) else {
            break;
        };
        let Some(moved) = above(lowest, highest) else {
            break; // no room above: leave the chord as it is
        };
        notes.remove(0);
        notes.push(moved);
    }
    notes
}

```


A chord of three notes has three positions, so turning it three times is the root position again: `times % notes.len()` skips the full turns. `.max(1)` keeps an empty list from dividing by zero, which would panic. The `let ... else` takes the two ends or leaves the loop. Unlike `invert`, this one does not refuse when a note cannot go higher: a stamp near the top of the keyboard still draws a chord, in the last inversion that fits.

Arpeggios: `cycle`, `step_by` and `zip`

Arpeggiate chords plays each chord's notes one after another, for as long as the chord lasted. The Tools menu picks the order (**up, down, up and down**) and how long each note is (**1 step, 2 steps, 1 beat**). The order is an enum:

```
#[derive(Debug, Clone, Copy, PartialEq, Eq, Default)]
pub enum ArpeggioOrder {
    #[default]
    Up,
    Down,
    UpDown,
}
```

`#[derive(Default)]` on an enum needs to be told which variant is the default, with `#[default]`. Each order knows which note of the chord comes next, counted from the lowest, forever:

```
fn indices(self, count: usize) -> impl Iterator<Item = usize> {
    let once: Vec<usize> = match self {
        ArpeggioOrder::Up => (0..count).collect(),
        ArpeggioOrder::Down => (0..count).rev().collect(),
        ArpeggioOrder::UpDown => (0..count)
            .chain((1..count.saturating_sub(1)).rev())
            .collect(),
    };
    // `cycle` repeats an iterator forever. It is lazy, so "forever"
    // costs nothing: the caller takes only as many as it needs.
    once.into_iter().cycle()
}
```

For three notes, up and down is 0, 1, 2, 1, and then again from 0: the top and bottom notes are played once on each turn, never twice in a row. `saturating_sub` stops at zero instead of going below it, so `count - 1` cannot underflow the `usize` for an empty chord. The return type `impl Iterator<Item = usize>` says "some iterator of `usize`" without naming its type, which here would be `Cycle<std::vec::IntoIter<usize>>`; callers only need to know they can iterate it.

```

pub fn arpeggiate(notes: &[NoteEvent], order: ArpeggioOrder, rate: usize)
-> Vec<NoteEvent> {
    let rate = rate.max(1);
    let chords = chords(notes);
    let mut result = Vec::new();
    for (index, chord) in chords.iter().enumerate() {
        if chord.len() < 2 {
            result.extend(chord);
            continue;
        }
        let start = chord[0].start;
        let mut end =
chord.iter().map(NoteEvent::end).max().unwrap_or(start + 1);
        if let Some(next) = chords.get(index + 1) {
            end = end.min(next[0].start);
        }
        // `step_by` takes every `rate`-th step; `zip` pairs each of those
        // steps with the next note of the order, and stops when the steps
        // run out (the order never does).
        for (step, pick) in
(start..end).step_by(rate).zip(order.indices(chord.len())) {
            result.push(NoteEvent {
                start: step,
                length: rate.min(end - step),
                ..chord[pick]
            });
        }
    }
    result
}

```

The arpeggio lasts until the chord's longest note ends, but stops where the next chord or single note starts, so it never plays over it. `map(NoteEvent::end)` passes the method as a function, as `to_vec` was above; `max` returns an `Option`, because an empty list has no largest item. `zip` stops when either side runs out, and the endless order never does, so the steps decide how many notes there are. Each note is built `..chord[pick]`, from the note of the chord it plays, so it keeps that note's velocity: a loud top note stays loud in the arpeggio. The last note is cut short if the chord ends in the middle of it (`rate.min(end - step)`).

`result.extend(chord)` adds a single note as it is. `chord` is a `&Vec<NoteEvent>`, and a `Vec` of a `Copy` type can be extended from references, copying each item.

Hearing what moved

After a tool that changes pitches, the piano roll plays the first chord of the result, so you hear where it went without starting the song:

```
if matches!(
    tool,
    Tool::Transpose(_) | Tool::InKey(_) | Tool::Invert { .. }
) && let Some(first) = tools::chords(&new).into_iter().next()
{
    actions.push(RollAction::Preview {
        channel: self.channel,
        pitches: first.iter().map(|note| note.pitch).collect(),
    });
}
```

`matches!` asks whether a value fits a pattern and gives a `bool`; `_` and `{ .. }` accept any value inside. It is a `let` chain again: the preview is sent only for those tools *and* if there is a first chord. `Preview` is one more action the app handles after drawing.

Testing it

Because the tools are plain functions, the tests are plain too: a few notes in, the expected notes out, no window. A helper makes a note from a MIDI number, a start and a length:

```
fn note(midi: u8, start: usize, length: usize) -> NoteEvent {
    NoteEvent::new(Note::new(midi), start, length)
}
```

and the tests read like the music:

```
#[test]
fn moving_in_key_changes_the_chord() {
    // C major one step up in C major is D minor: D, F, A.
    let c_major = [note(60, 0, 4), note(64, 0, 4), note(67, 0, 4)];
    let up = transpose_in_key(&c_major, C_MAJOR, 1).unwrap();
    assert_eq!(pitches(&up), [62, 65, 69]);
    // Two steps down from C, E, G: A, C, E (A minor).
    let down = transpose_in_key(&c_major, C_MAJOR, -2).unwrap();
    assert_eq!(pitches(&down), [57, 60, 64]);
    // ...
}
```

The others check that one note too high spoils the whole transposition, that a shift stays inside the pattern, that eighth-note triplets quantize to steps 0, 1 and 3, that snapping moves only the notes outside the key (and the lower note wins a tie), that inverting three times is the root position again, that an arpeggio stops at the next chord and keeps each note's velocity. The tests of `change_key`, the function the **Key** menus use, which lives in the same file, are in [Scales, modes and keys](#).

Exercises on the tools are in [Exercises: music theory in code](#).

Hearing harmony: chords and keys from notes

Files: `crates/lineup-core/src/harmony/hearing.rs`, `crates/lineup-core/src/harmony/key_finding.rs`, `crates/lineup-core/src/library/listen.rs`, `crates/lineup-core/src/project.rs`, `crates/lineup-app/src/app/key_finder.rs` and `crates/lineup-core/tests/chord_reading.rs`

A pattern stores notes, not chords, and an imported MIDI file or a song you played in has no chords written anywhere. Yet many parts of the app need the harmony: the chord sheet's **Write the chords the song plays**, the **All** row of the piano roll's chord lane, the harmony key that lights up as the song plays, the voice-leading check, a style that plays a whole song again in another rhythm, and the key of a MIDI file without key signatures. All of them listen to the notes the way a musician does by ear. This chapter is about how: first chords, then keys.

`analyze`, in [Chords: building, naming and inversions](#), names the notes held at one moment, which is right for a chord held down on a keyboard. Music that moves needs more.

The music, briefly

Naming the notes of one moment goes wrong in five common ways:

- An **arpeggio** plays its chord one note at a time: C, then E, then G. At any moment only one note sounds, and one note names no chord.
- A **tune** passes through notes outside the chord. Under a C major chord, a melody on D makes the moment C, D, E, G, which reads as Cadd9 though the harmony is plain C.
- A **walking bass** steps between chords: C, B $\flat$, A on the way from C to A7. The B $\flat$ under C, E and G makes C7 for a moment, though no one plays C7.
- An orchestra can play **two chords at once**, one over the other: a **polychord**. In the Augurs of Spring, from Stravinsky's *Rite of Spring*, the low strings play F $\flat$ major (F $\flat$, A $\flat$, C $\flat$) while E $\flat$ 7 (E $\flat$, G, B $\flat$, D $\flat$) sounds above. In *Petrushka* the clarinets play C major over F $\sharp$ major. No single chord symbol holds either one.
- Some harmonies are **not stacks of thirds** at all. *Petrushka*'s fair opens on a flute figure on D, E, G and A, which stack up in fourths: E, A, D, G. Calling that a chord forces a name on it.

A **key** is heard the same way, over time. In C major, C, E and G come back most, then the other white keys, and the black keys hardly at all. A minor uses the same white keys, but A, C and E lead, and G# turns up at every cadence. A piece **modulates** when it moves to a new key and stays there; a single chord from another key, such as D7 leading to G in a song in C major (a *secondary dominant*), only *tonicizes* G for a moment and is no change of key.

Listening to a stretch, not a moment

The chord hearing listens to a **stretch** of music, a beat, half a bar or a bar, and weighs every note by how long it sounds there. A note heard is a `HeardNote`, timed in steps from the start of the stretch:

```
pub struct HeardNote {
    pub pitch: Note,
    /// When it starts, in steps from the start of the stretch: below 0
    /// for a note that started before it.
    pub start: f32,
    /// When it ends, in steps from the start of the stretch: beyond its
    /// length for a note that goes on after it.
    pub end: f32,
    /// Which part plays it, such as a channel: an instrument's notes are
    /// a layer of their own when two chords sound at once.
    pub part: usize,
}
```

The times are `f32`, not whole steps, because a note can start between steps (its nudge, see [The project: patterns, channels, tracks, clips and markers](#)). Notes can also lie wholly outside the stretch. Those weigh nothing, but they tell what each part is doing around it, as the next section shows. `Project::heard_notes` gathers the notes of every channel that plays notes (synths and MIDI Out channels that are not muted; drums and samplers do not) between two steps of the song.

Adding up how long each of the twelve pitch classes sounds gives a **profile**:

```
struct Profile {
    weight: [f32; 12],
    total: f32,
    /// How long the pitch class heard longest sounds.
    loudest: f32,
}
```

A C major arpeggio over a bar is C, E and G each sounding a third of it: the same proportions as a held chord, so the arpeggio is heard as its chord. That one idea fixes

the first problem in the list. The others need to know what each part is doing.

What each part does

How much a note weighs depends on the part that plays it. `weights` tells four roles apart:

Role	How it is told	What its notes weigh
Chords	three different notes or more sound together in it, or it plays chords one note at a time	in full
Bass line	one note at a time, all below the chords	in full if the chords have the note; 0.15 if not, a passing note
Tune	any other part, when some part plays chords	a quarter
Layer	a part that would be a tune, but whose notes are three or more of one triad or seventh chord, none of them in the chords	in full

The end of `weights` applies the table, note by note:

```
let weight = if tune {  
  w * TUNE  
} else if lowest_chord.is_some()  
  && below  
  && !chord_notes.contains(note.pitch.pitch_class())  
{  
  w * PASSING_BASS  
} else {  
  w  
};
```

`w` is how long the note sounds inside the stretch. When no part plays chords, as in an orchestra where every instrument plays one line, `lowest_chord` is `None`, nobody is a tune and every line counts in full.

The **layer** row is there for the Rite of Spring. Its ostinato D_b , B_b , E_b , B_b is one line, but it spells E_b7 against the C major and E minor arpeggios below it: a harmony of its own, not a tune.

Two more questions need the bars around the stretch, since a single beat cannot answer them:

- Does a part **move its chords along with a tune**, every voice the same way by a step or a third, as in the last statements of Ravel's *Boléro*? Then its chords colour the tune and the harmony is what goes on under them, so the part counts as a tune. That is `moves_in_parallel`; two chords are not enough to show parallel motion, but a phrase is.
- Does a part **play chords one note at a time**, as an Alberti bass or the triplets of the *Moonlight* Sonata do? Short notes, nearly all leaps of a minor third or more, on three pitch classes at least: `breaks_chords`. Then the part plays chords as much as one striking them does.

A score for every chord

Every chord, twelve roots times every quality the app knows, gets a score for how well it explains the profile:


```

fn chord_fit_with(chord: Chord, profile: &Profile, also: PitchSet) ->
(f32, bool) {
    if profile.total <= 0.0 {
        return (f32::NEG_INFINITY, false);
    }
    let explained = PitchSet::from_bits(chord.pitch_set().bits() |
also.bits());
    let inside: f32 = explained.iter().map(|p| profile.of(p)).sum();
    let outside = profile.total - inside;
    let full = PRESENT * profile.loudest;
    let mut missing = 0.0;
    let mut core = true;
    for (&interval, &tone) in
chord.quality.intervals().iter().zip(chord.quality.tones()) {
        let pitch_class = chord.root.transpose(i32::from(interval));
        let presence = (profile.of(pitch_class) / full).min(1.0);
        let loss = if tone == 5 && interval % 12 == 7 {
            MISSING_FIFTH
        } else {
            MISSING
        };
        missing += loss * (1.0 - presence);
        if matches!(tone, 1..=4) && presence < 0.5 {
            core = false;
        }
    }
    let extra = chord.quality.intervals().len().saturating_sub(3) as f32;
    let fit = (inside - OUTSIDE * outside) / profile.total - missing -
SIZE * extra;
    (fit, core)
}

```

- `inside` is how much of the sound the chord explains, and `outside` what it leaves out, which counts one and a half times as much (`OUTSIDE = 1.5`): a wrong note is stronger evidence than a right one. Divided by the total, a chord that explains everything scores 1.
- `missing` is a small loss for each note of the chord that is hardly heard, 0.15 for a missing note and only 0.05 for a missing perfect fifth, which players leave out freely. A note heard at least 35% as long as the loudest (`PRESENT`) counts as fully there.
- `extra` counts the notes beyond a triad, 0.06 each. A thirteenth chord holds nearly every note of a scale, so without this cost it would explain anything; it wins only when its notes really sound.
- `core` says whether the root and the third (or the note that stands for it in a sus chord) are really heard. A chord without them is passed over.

`PitchSet` stores a set of pitch classes as the bits of one `u16` (see [Pitch classes, notes and spelling](#)). `|` on the bits is the union of two sets, so `explained` is “the chord’s

notes and those of `also` in one instruction. `also` is empty for a single chord; in a polychord it holds the other chord's notes.

The bass adds the last touch:

```
fn bass_fit(chord: Chord, bass: Option<PitchClass>) -> f32 {
  match bass.map(|bass| (bass, chord.tone_of(bass))) {
    Some((bass, _)) if bass == chord.root => BASS_ROOT,
    Some((_, Some(3 | 5 | 7))) | None => 0.0,
    Some(_) => -BASS_OUTSIDE,
  }
}
```

A root in the bass earns 0.06; a third, fifth or seventh is an inversion and costs nothing; any other note costs 0.06. The middle arm shows two pattern tools: `3 | 5 | 7` inside `Some(...)` matches any of three values, and `| None` joins a second whole pattern to the arm.

A walking bass, worked through

A piano holds C, E and G for two beats (eight steps) while the bass walks C, then B $\flat$, on its way down to A for the next chord. The bass plays below the chords and B $\flat$ is not one of their notes, so it weighs $4 \times 0.15 = 0.6$. The profile is C 12 (piano and bass), E 8, G 8 and B $\flat$ 0.6, 28.6 in all; `loudest` is capped at the stretch's length, 8, so a note doubled in two parts does not make the others look faint.

- **C** explains everything but the B $\flat$: $(28 - 1.5 \times 0.6) / 28.6 = 0.948$, nothing missing, plus 0.06 for its root in the bass: **1.008**.
- **C7** explains everything, 1.0. But its B $\flat$ is heard for 0.6 of the 2.8 that counts as fully there, which costs $0.15 \times (1 - 0.21) = 0.12$, and its fourth note costs 0.06. With the bass bonus: **0.882**.

C wins, as a chart would say, and C7 is not even kept as a second reading (that needs to be within 0.12 of the best). Had the passing note weighed 0.35, C would still win, by 0.941 to 0.925, but C7 would show as a close alternative.

Keeping the best three

Only the three best chords become readings, so there is no need to sort all of them. A short vector stays sorted as chords arrive:

```
let at = found.partition_point(|&(_, other): &(Chord, f32)| other >= fit);
if at < MOST_READINGS {
    found.insert(at, (chord, fit));
    found.truncate(MOST_READINGS);
}
```

`partition_point` does a binary search for the first place where the condition turns false: the first chord that fits worse than the new one. A chord that fits exactly as well goes after those already there, so the result is the same as a stable sort, at a fraction of the work.

Two chords at once

If no single chord fits well (a fit under 0.7, `ONE_CHORD`), the hearing tries two.

`best_polychord` splits the notes into a lower and an upper layer in every way that makes sense:

- **by register:** everything below a pitch is the lower layer. A split that would cut through a line is skipped, since a line running from A up to G is an arpeggio of A7, not the top of G major;
- **by instrument:** the parts sorted from the lowest sounding to the highest, cut between any two of them.

Each layer must make a major or minor triad, or a major, minor or dominant seventh chord, with three of its notes heard and its root among them. Two bigger chords at once would explain any notes at all. Then the pair must pass four checks:

1. The roots differ, and the upper chord adds at least two notes of its own. C over A minor is just Am7.
2. The two chords do not make **one key** between them. F major over G major uses only the notes of C major: that is G eleventh, one harmony in C, not two keys at once.
3. Both chords **sound together** through at least 80% of the time anything sounds in the stretch (`covered`). C major and then G7 is a progression, not a polychord. The ear holds the notes of an arpeggio, or of a staccato chord struck again and again, so a gap of two steps or less counts as sounding.
4. The pair, less 0.2 for needing two chords (`POLYCHORD`), scores better than the best single chord.

The second check is a few lines:

```
fn in_one_key(notes: PitchSet) -> bool {
    let major = PitchSet::from_semitones(&[0, 2, 4, 5, 7, 9, 11]);
    (0..12).any(|tonic| notes.is_subset_of(major.transpose(tonic)))
}
```

`transpose` rotates the bits, so this tries all twelve major scales, and their modes with them. Why only major scales? Because the Augurs chord itself fits a **harmonic minor** scale. Its seven notes, F $\flat$, G, A $\flat$, B $\flat$, C $\flat$, D $\flat$ and E $\flat$, are exactly A $\flat$ harmonic minor. Had the check counted harmonic minor as one key, the chord would have been rejected.

The Petrushka chord passes the same way. C major and F $\sharp$ major lie a tritone apart and share no note, and between them they hold six notes, C, C $\sharp$, E, F $\sharp$, G and A $\sharp$, that no single major scale holds. Both triads come from the **octatonic** scale, C, D $\flat$, E $\flat$, E, F $\sharp$, G, A, B $\flat$, which alternates half and whole steps.

A set when no chord fits

Last, the notes are scored as a **pitch-class set** (see [Pitch-class sets, and one harmony with four names](#)). A set explains every note heard enough to count, so it always fits, but it pays 0.35 for giving up on a chord name. A stack of fourths pays only 0.1, because since Debussy it has been a harmony of its own, as plain to the ear as a triad:

```
/// The sets that are stacks of fourths: C F Bb (3-9, the same notes as a
/// sus chord), C F Bb Eb (4-23) and C F Bb Eb Ab (5-35, the pentatonic
/// scale), as their prime forms.
const QUARTAL_SETS: [&[u8]; 3] = [&[0, 2, 7], &[0, 2, 5, 7], &[0, 2, 4, 7,
9]];
```

The flute's D, E, G and A at the start of *Petrushka*'s fair is 4-23, so it is named as that set, instead of a forced chord such as D6sus2.

The best reading, and the ones close behind

A `Hearing` keeps every reading within 0.12 of the best (`CLOSE`), three at most:

```
pub struct Hearing {
    readings: Vec<Reading>,
    /// The pitch classes heard enough to be part of the harmony, without
    /// the passing notes.
    notes: PitchSet,
    /// How long each pitch class sounds, weighed.
    profile: [f32; 12],
}
```

`best()` is the first reading, or `Harmony::Silence` when there is none, and `alternatives()` the rest:

```
pub fn alternatives(&self) -> &[Reading] {
    self.readings.get(1..).unwrap_or_default()
}
```

`get(1..)` returns `None` for an empty vector, where `&self.readings[1..]` would panic, and `unwrap_or_default` turns that `None` into an empty slice, the default of `&[Reading]`.

The fields are private, so a `Hearing` can only be made by hearing: nothing outside the module can build one whose readings are out of order. The app shows the readings the way it shows its other cues, the most likely brightest and the others fainter:

- Next to the harmony keys' hint, a line says what is heard when it is not simply one chord: "Heard: Eb7 | Fb, two chords at once", "Heard: the notes 4-23", or a chord followed by "or" and its close rivals. On the harmony keys, each chord of a polychord lights its key.
- The piano roll's status line, for the step under the pointer, adds the other readings in brackets: "(or C6)".

One last rule protects a tune alone. Für Elise opens with E, D#, E, D#, E, B, D, C and nothing under them: a tune, not a chord, though added up its notes would earn some chord name. So when a single line plays alone and its best reading fits less than 0.85 (`ALONE`), the hearing names no harmony at all, and the chord before it goes on.

Cutting a bar into chords

Write the chords the song plays fills the chord sheet from the notes. It also runs by itself when a band style arrives in a song whose sheet is empty, since the band plays from the sheet. It takes the song a bar at a time and asks: where should this bar be cut? A bar of 4/4 has beat lines at 0, 1, 2, 3 and 4, and any two of them make a stretch:

ten stretches, from single beats to the whole bar. The best way to cut is found by **dynamic programming**:

```
// best[i]: the best score for the beats before line i, where its
// last stretch starts, and how that stretch is heard.
let mut best: Vec<(f32, usize, Option<Hearing>> = vec![(0.0, 0, None)];
for i in 1..lines.len() {
    let mut here = (f32::NEG_INFINITY, 0, None);
    for j in 0..i {
        let hearing = heard(lines[j], lines[i]);
        let cut = match lines[j] {
            line if line == bar.start => 0.0,
            line if line == half => CUT_HALF_BAR,
            _ => CUT_BEAT,
        };
        let total = best[j].0 + worth(&hearing) * (i - j) as f32 - cut;
        if total > here.0 {
            here = (total, j, Some(hearing));
        }
    }
    best.push(here);
}
```

The idea: the best way to cut the first `i` beats ends with some last stretch, from line `j` to line `i`. Whatever `j` is, the cuts before it must be the best way to cut the first `j` beats, or there would be a better answer. So `best[i]` is worked out from the `best[j]` before it, which are already known, trying every `j`. A bar of four beats can be cut in only $2^3 = 8$ ways, but the same stretch turns up in many of them, and this way each of the ten stretches is heard once. The library's listener, below, cuts a whole pattern the same way, where trying every way would be hopeless: a pattern of 64 beats can be cut in 2^{63} ways, and the loop asks about $64 \times 64 / 2$ questions.

Each stretch earns its fit once per beat it covers (`worth`), and each cut costs a little: 0.08 halfway through the bar, 0.12 on any other beat. So a bar is split only where two chords really fit better than one, and more easily in the middle than on a weak beat, as chords change in real music. `f32::NEG_INFINITY` is smaller than every other float, so the first real total always beats it, with no special case. Walking back from the end through the `j` s gives the cuts, last first, and `reverse` puts them in order.

`worth` gives silence a full score, and a single line a little less:

```
fn worth(hearing: &Hearing) -> f32 {
    match hearing.best() {
        Harmony::Silence => 1.0,
        Harmony::Single(_) | Harmony::Interval { .. } => 0.8,
        _ => hearing.fit(),
    }
}
```

A beat of rest, or of a melody alone, then joins the stretch around it rather than being cut off, and writes nothing: the chord before goes on.

A polychord also **goes on** while its notes do. In *Petrushka's* Room, the F# major half of the chord rests for a bar while C major goes on. Heard alone, that bar is C. But C major and F# major still explain nearly all of it, so where nothing else fits well the sheet keeps C|F# instead of jumping to C and back (`Hearing::goes_on`).

Hearing fast

Writing the sheet hears ten stretches of every bar of 4/4, and the tests write the sheets of every demo, so speed matters. Three things keep it fast.

Do shared work once. Which parts move in parallel, and which play broken chords, depends on the bar and the bars around it, not on the stretch. A small struct works it out once and hears the stretches from it:

```
pub struct Listening<'a> {
    notes: &'a [HeardNote],
    /// The parts that move their chords along with a tune.
    planing: Vec<usize>,
    /// The parts that play chords one note at a time.
    broken: Vec<usize>,
}
```

'a is a **lifetime**. `Listening` borrows the notes instead of copying them, and 'a tells the compiler that a `Listening` must not outlive the slice it borrows. `hear(start, end)` then picks the notes of one stretch and retimes them from its start. The piano roll does the same for its All row: it gathers each bar's notes once and hears its beats from them.

Don't ask a question per note that could be asked per part. Whether a part is a tune depends on all of its notes. Asked again for every note of the stretch, that is n^2 work; `weights` asks once per part, keeps the answers in `roles`, and looks each note's part up.

Don't hear the same bar twice. Songs repeat. A bar played again exactly as before, between the same bars, is heard the same way, so `write_heard_chords` keeps a `HashMap` from each bar's beat lines and notes to how it was cut:

```
type HeardBars = HashMap<(Vec<usize>, Vec<(i64, i64, u8, usize)>),  
Vec<(usize, Hearing)>>;
```

The notes go into the key as whole numbers, `(start, end, pitch, part)`, since an `f32` cannot be a key: it has no `Hash` and no `Eq`, because `NaN` is not equal to itself. `type` gives the long type a short name; it is the same type, not a new one.

To find what is slow, measure. `valgrind --tool=callgrind` runs a program on a simulated processor and counts every instruction, per function and per line:

```
CARGO_PROFILE_RELEASE_DEBUG=true cargo test --release -p lineup-core --no-  
run  
valgrind --tool=callgrind ./target/release/deps/<test binary> --ignored  
one  
callgrind_annotate --inclusive=yes callgrind.out.<pid>
```

It points at the function, and the line, where the time goes; a question asked per note that could be asked per part is the kind of thing it finds.

Writing it down

Each stretch becomes a chord marker where the harmony changes, and again where a section starts, as a chart writes the chord again at the top of each part. A chord is spelled in the key of the song where it plays, with the band's lowest note after the slash: piano chords voiced C, E, A over a bass playing A are Am, not Am/C. A polychord is written `upper | lower`, `Eb7 | Fb`, and a set by its name, `4-23 T2` (see [The chord sheet: model, layout and text](#)). In a key that is a set or a twelve-tone row, where chords have no function, each beat is written as the set its notes make instead, or as a micromode (see [Micromodes](#)).

A second ear, for the band's own patterns

The library hears chords too, in a simpler setting. To play a song in a new rhythm, or to let the band follow the chords of a pattern, it needs the chords of one **chord channel** over one **bass line**, in whole steps: the patterns a style writes.

`library/listen.rs` does that with the same two ideas, a score for every chord and

dynamic programming for the cuts, tuned so that every style of the library, however it plays its chords, is heard as its chart.

Its scores need “how many steps does each pitch class sound between step `a` and step `b`”, for thousands of stretches. So it adds up once, from the start:

```
struct Heard {
    sums: Vec<f32; 12>,
    busy: Vec<f32>,
    quiet: Vec<f32>,
    /// The pitch class of the lowest bass note that starts at each step,
    /// if any.
    bass: Vec<Option<PitchClass>>,
}
```

`sums[s][pc]` is how many of the steps *before* `s` have pitch class `pc` sounding. The time between `a` and `b` is then one subtraction, however long the stretch:

```
fn between(&self, start: usize, end: usize) -> Stretch {
    let mut heard = [0.0; 12];
    for (pc, time) in heard.iter_mut().enumerate() {
        *time = self.sums[end][pc] - self.sums[start][pc];
    }
    // ...
}
```

This is a **prefix sum**, and it turns up wherever a program asks “how much in this range” many times. Arrays of a fixed size are `Copy` when their items are, so building the sums with `let mut next = heard.sums[heard.sums.len() - 1];` copies the last row, and adding to `next` leaves the row in the vector untouched.

After the cuts, neighbours are joined. Two stretches with the same chord become one, and so do a C and a C7 a beat later, since an arpeggio shows its notes one by one:

```
match joined.last_mut() {
    Some(last) if last.chord == stretch.chord => last.length +=
stretch.length,
    // An arpeggio shows its notes one by one: a C that turns into
    // C7 a beat later is one C7 still unfolding. A change on a
    // bar line or half way through the bar is taken as meant.
    Some(last) if !half_bar(stretch.start) && grows(last.chord,
stretch.chord) => {
        last.chord = bigger(last.chord, stretch.chord);
        last.length += stretch.length;
    }
    _ => joined.push(stretch),
}
```

`last_mut()` gives an `Option<&mut HeardChord>`, a reference to change the last stretch through, so it grows in place. The arms are tried in order with their **match guards** (the `if` after a pattern); a false guard moves on to the next arm, and `_` catches the rest. What the band does with the chords it hears is in [The library: styles and rhythms as data](#) and [How the band plays a chord sheet](#).

Finding the key

The second half of the chapter is the key. 🔍 **Find the key**, under the harmony keys, shows the key the music sounds like it is in, with its closest rivals fainter beside it, and what the chord playing does in each. It is an experiment, so it is off until you turn it on. The work is in `harmony/key_finding.rs`.

What a key sounds like: profiles

Carol Krumhansl and Edward Kessler (1982) played listeners a scale and then a single note, and asked how well the note fitted. The answers make a **key profile**: twelve numbers, how much each note above the tonic belongs to the key. Later studies counted notes in real scores instead. Ferrous Lineup uses the profiles of Albrecht and Shanahan (2013), counted over 490 classical pieces, which named the keys of the demos best of those tried:

note above the tonic	1	$\flat 2$	2	$\flat 3$	3	4	$\sharp 4$	5	$\flat 6$
major	.24	.01	.11	.01	.14	.09	.02	.21	.01
minor	.22	.01	.10	.12	.02	.10	.01	.21	.06

The tonic and the fifth stand out in both; the third tells major (3) from minor ($\flat 3$). In minor, $\flat 7$ and 7 are close: the natural seventh of the scale and the raised one of harmonic minor both turn up.

Comparing: the Krumhansl-Schmuckler method

To find the key of some music:

1. Add up how long each of the twelve pitch classes sounds, in any octave: the music's **chroma**, C first. (It is the same kind of twelve numbers as the chord hearing's profile.)
2. Turn each profile to start on each of the twelve tonics: 24 keys.
3. Compare the chroma with each of the 24 using their **correlation**: 1 when they rise and fall together, 0 when unrelated, -1 when opposite.
4. The key with the highest correlation wins.

The correlation is two steps: move both lists to an average of 0 and a length of 1, then take their dot product.

```
fn normalized(values: &Chroma) -> Option<Chroma> {
  let mean = values.iter().sum::<f32>() / 12.0;
  let centered = values.map(|v| v - mean);
  let length = centered.iter().map(|v| v * v).sum::<f32>().sqrt();
  (length > 1e-9).then(|| centered.map(|v| v / length))
}
```

`Chroma` is `[f32; 12]`, and arrays have a `map` of their own that returns an array of the same size, so no `Vec` is made. `Option` handles silence: when nothing sounds, all twelve numbers are equal, the length is 0, and there is no key to find, so the function returns `None` instead of dividing by zero. `then` is the lazy cousin of `then_some`: it runs the closure only when the condition holds, so the division never happens for silence.

Turning a profile to another tonic uses `std::array::from_fn`, which builds an array by calling a closure for each index:

```
// The profile counts from the tonic; turn it to count from C.
let turned: Chroma = std::array::from_fn(|c| profile[(c + 12 - tonic) % 12]);
```

`key_fits` returns all 24 keys with their correlation, best first. The MIDI import uses it to check a file's key signature against its notes (see [MIDI: messages, devices, recording and files](#)).

Sure or not sure

The correlations alone do not say how sure the answer is. C major and A minor have the same notes, so a tune on the white keys can fit both almost equally well. `ranked` turns the 24 correlations into **shares** that add up to 1, a *softmax*:

```
let best = fits.iter().copied().fold(f32::MIN, f32::max);
let weights: Vec<f32> = fits
    .iter()
    .map(|&fit| ((fit - best) * SHARE_SHARPNESS).exp())
    .collect();
let total: f32 = weights.iter().sum();
```

With `SHARE_SHARPNESS` at 12, a fit better by 0.05 gets a share $e^{(12 \times 0.05)} \approx 1.8$ times larger. Subtracting `best` first keeps every exponent at 0 or below, so `exp` can never overflow. A key is **clear** when its share is at least twice the second's:

```
fn clear(guesses: &[KeyGuess]) -> bool {
    match guesses {
        [] => false,
        [_] => true,
        [best, second, ..] => best.share >= CLEAR_RATIO * second.share,
    }
}
```

These are **slice patterns**: `[]` matches an empty slice, `[_]` one of exactly one item, and `[best, second, ..]` names the first two of a longer one, with `..` for the rest. When the key is not clear, the next keys whose share is at least 40% of the best are its **alternatives**, two at most, and the app shows them fainter.

E Dorian has the same notes as A minor and G major, so the profiles, which only know major and minor, cannot pick its home note with confidence, and the app says so: A minor, or E minor, or A major. The real tonic is among the alternatives, which is the honest answer.

Where the key changes

A song that modulates has no single chroma. So the music is cut into bars, and each bar gets a score for each key. One bar is too little to go on, though: a bar holding only D7 fits G major better than C major, even in the middle of a C major song. So each bar is scored over the bars around it, itself counted three times, its neighbours twice, the bars two away once.

Then a key change **costs** a penalty, 1.5 in units of correlation, and the **Viterbi algorithm** finds the sequence of keys with the best total score. For each bar and each key it keeps only the best way of arriving there: staying in that key, or changing from the best key so far and paying the penalty.

```

for (i, here) in windowed.iter().enumerate().skip(1) {
    let (leader, leader_score) = best_of(&best);
    let mut next = vec![0.0; count];
    for k in 0..count {
        let (previous, total) = if best[k] >= leader_score - CHANGE_COST {
            (k, best[k])
        } else {
            (leader, leader_score - CHANGE_COST)
        };
        from[i][k] = previous;
        next[k] = total + here[k];
    }
    best = next;
}

```

`from[i][k]` remembers where the best path into key `k` at bar `i` came from, and at the end the code walks back through it to read off the path. It is dynamic programming again, over keys instead of cuts. A new key wins only when it fits better for long enough to pay back the penalty, which is exactly “a change only once the new key holds”: one borrowed chord is not a modulation.

Scoring over neighbouring bars blurs *where* a change happens by a bar or two. So each change is then moved, within three bars, to the bar where the bars’ own notes switch from fitting the old key to fitting the new one. In the Habanera from *Carmen* it lands on bar 10, where Bizet’s D minor turns to D major.

`Project::heard_keys` does this for the whole song: synths and MIDI Out channels that are not muted count, drums and samplers (such as an imported recording) do not, and each note counts for as long as it sounds in each bar. It returns `KeySpan`s, each with a start, an end and every key ranked. **Mark on the playlist**, next to the key heard, turns them into key markers (see [Scales, modes and keys](#)): no note moves, and Undo takes them back. It is the way to give an imported MIDI file without key signatures the key changes it really has.

While you play

For chords played on a MIDI or typing keyboard there is no score to look ahead in, only the bars played so far. `KeyListener` keeps a chroma of the past that fades:

```

/// A bar ended: it joins the past, and the key may change.
pub fn end_bar(&mut self) {
    if normalized(&self.bar).is_none() {
        return;
    }
    let heard = self.heard();
    self.past = heard.map(|v| v * LISTENER_MEMORY);
    self.bar = [0.0; 12];
    // ...
}

```

Each bar, everything heard before counts 0.84 times as much, so a bar from four bars ago counts about half ($0.84^4 \approx 0.5$). A new key takes over only after it has led for two bars in a row. The listener does not start from nothing either:

`KeyListener::starting_in` fills the past with music shaped like the key around it, the key heard in the song's notes there or the project's key, so the first chords you play are heard in its light. Play C, F and G7 over an E Dorian song and they are first heard against A minor; a few bars later the listener settles on C major, clear.

The app shows the key from the most immediate source it has (see `key_finder.rs`): what you play, if you played in the last four bars; otherwise the keys heard in an imported recording, or in the song's notes at the playhead; in pattern mode, the selected pattern's notes. The line says which: "Sounds in what you play:", "in the notes here", "in this pattern". How a recording's keys are heard is in [Importing a recording \(experimental\)](#).

What a chord is in each key

The same chord does a different job in each key it could belong to, which is why the alternatives matter. `read_chord` works it out with what the app already knows about keys: the numeral, the function, a dominant seventh heard as the dominant of the chord it pulls to, and chords borrowed from the parallel key (see [Roman numerals, the harmony keys and chord functions](#)). G7 reads:

key	reading	what it does
C major	V7, dominant	pulls home to C
A minor	$\flat VII7 = V7/$ $\flat III$	the dominant of C, the relative major
G major	$I7 = V7/IV$	the home chord with a flat seventh, pulling to C, the IV

Numerals count along the major scale of the tonic, as everywhere in Ferrous Lineup, so G in A minor is $\flat$ VII.

How well it hears

The tests in `tests/chord_reading.rs` check bars whose harmony is known from the scores: the Augurs chord, the Petrushka chord in *Petrushka's* Room, and the fourths at the start of the fair. A report, which is an ignored test, compares the chords heard from the notes with the chart of every demo that has one, beat by beat:

```
cargo test --release -p lineup-core --test chord_reading -- --ignored --nocapture
```

`tests/key_finding.rs` has a report of the same kind for keys, against the keys and key markers the demos have from their scores:

```
cargo test --release -p lineup-core --test key_finding report -- --ignored --nocapture
```

Measured on 30 October 2026:

Chords, 330 charted demos, 60,322 beats	
Exactly the chart's chord	84.0%
A compatible chord (same notes, or same root with notes added or left out)	91.1%
The right root	92.1%

Keys, from the notes	
Bars in the right key	87.0% of 18,828
Key changes in the scores found within a bar	132 of 315
Bars shown as clear that are right	13,900 of 16,153 (86%)

Each report also lists the demos it gets most wrong: the place to look next. For keys, most misses are of three kinds:

- **Short visits.** A change that lasts four to eight bars, often to the dominant, is hard to tell from a passage that only tonicizes it, and many musicians would read those bars as tonicizations too.
- **Relative keys.** Same notes, another home: a song in $\flat$ B major can come out as G minor.

- **Modes**, as with E Dorian above.

Some “changes that are not there” are real, too: the scores the demos come from mark only the bigger key changes.

Testing it

The unit tests in `hearing.rs` build a few notes by hand and check what is heard. An arpeggio, one note per step, is its chord:

```
#[test]
fn an_arpeggio_is_its_chord() {
    // A minor broken up, one note per step: no two notes ever sound
    // together.
    let notes = [
        held(45, 0.0, 1.0),
        held(52, 1.0, 2.0),
        held(57, 2.0, 3.0),
        held(60, 3.0, 4.0),
    ];
    assert_eq!(chord_of(&hear(&notes, 4.0)).unwrap(), "Am");
}
```

`held` makes a note of part 0, and `chord_of` names the best reading. `unwrap` is fine in a test: if there is no name, the test should fail right there. The others check that a held chord is itself, that a tune’s passing notes leave the chord alone, that the Augurs chord is Eb7 over Fb and the Petrushka chord two arpeggios a tritone apart, that a stack of fourths is a set, and that nothing heard is silence. `key_finding.rs` checks that a cadence names its key, that the shares add up to one, that the pentatonic scale alone is not clear, that one borrowed bar is not a modulation and that the listener changes key once the new one holds; `tests/key_finding.rs` checks that most bars of the demos are heard in their key and that clear modulations, such as the Habanera’s, are found on their bar.

Exercises on hearing are in [Exercises: music theory in code](#).

Pitch-class sets, and one harmony with four names

Files: `crates/lineup-core/src/harmony/set_class.rs`, `crates/lineup-core/src/harmony/pitch_set.rs`, `crates/lineup-core/src/chord_sheet/notation.rs`, `crates/lineup-app/src/twelve_tone.rs` and `crates/lineup-app/src/chord_sheet/notation.rs`

Chord names work for the chords of tonal music. Much music of the 20th century uses groups of notes that have no chord name, such as C, C# and F#. **Pitch-class set theory**, worked out by Milton Babbitt, Allen Forte, John Rahn and others, gives every group of notes a name. In Ferrous Lineup a set is one more way to *write* a harmony, next to chord symbols, Roman numerals and micromodes: switching to sets never moves a note. This chapter is about the theory, the few lines of bit arithmetic that compute it, and the places the app uses it: the set browser, a key shown as a set, and the chord sheet's four notations.

The music, briefly

Pitch classes as numbers. C is 0, C# is 1, and so on up to B, 11. Octaves and spellings do not count: every C is 0, and C# and D $\flat$ are both 1. A **set** is a choice of some of the twelve; there are $2^{12} = 4,096$ of them, from the empty set to all twelve.

Moving a set. *Transposing* (Tn) moves every note up n semitones: C E G up 3 is E $\flat$ G B $\flat$. *Inverting* (I) turns every interval upside down: C E G (up 4, then up 3) becomes C A $\flat$ F (down 4, then down 3), which is F minor. Upside down, a major triad is a minor triad.

Set classes. A set together with all its transpositions and inversions is a **set class**. All 24 major and minor triads are one class. The 4,096 sets fall into **224 classes**.

Normal form is the most compact way to write a set. Written around a circle, a set can start on any of its notes; start on each in turn and measure the span from the first note to the last. Take C major seven, C E G B:

starting on	notes	span	to the note before last
C	C E G B	11	
E	E G B C	8	7
G	G B C E	9	

starting on	notes	span	to the note before last
B	B C E G	8	5

E and B tie with a span of 8, so compare the span to the note before last: B to E is 5, E to B is 7. The normal form is **B C E G**. (This tie-break, comparing from the end, is John Rahn's; Allen Forte compared from the start, and the two disagree for six classes. Ferrous Lineup uses Rahn's, as most textbooks now do.)

Prime form goes one step further: take the normal forms of the set *and of its inversion*, move both to start on 0, and keep the more compact one. For Cmaj7 it is **[0158]**: B C E G moved down to start on 0 is 0 1 5 8. Every major seventh chord, on any root, has the prime form [0158].

Forte numbers list the classes by size, then in Forte's order: [0158] is **4-20**, the twentieth class of four notes. The triads are 3-11, the diminished seventh chord 4-28, the major scale 7-35, the whole-tone scale 6-35. Books on 20th-century music use these numbers everywhere.

The interval vector. Every pair of notes in a set makes an interval. Counting them by size gives six numbers: how many minor seconds, major seconds, minor thirds, major thirds, fourths and tritones. A fifth counts as a fourth (it is a fourth upside down) and a major seventh as a minor second, so six sizes are enough. The vector is a fair summary of what a set *sounds* like. For C E G B:

```
C-E major third   C-G fourth   C-B minor second
E-G minor third   E-B fourth   G-B major third
<1 0 1 2 2 0>, written <101220>
```

Two curiosities follow from it:

- **Z-related** classes have the same interval vector without being the same class. 4-Z15 [0146] and 4-Z29 [0137] each hold one interval of every size, **<111111>** (they are the two **all-interval tetrachords**), yet neither can be moved onto the other. The **Z** in their names marks them.
- **Complements**: the notes a set leaves out make another set. The white keys (7-35) leave out the black keys, the pentatonic scale (5-35). A set of six notes and its complement always have the same interval vector.

Set theory in twelve bits

A **PitchSet** is twelve bits of a **u16**, bit 0 for C up to bit 11 for B, and transposing is rotating the bits (see [Pitch classes, notes and spelling](#)). The rest of set theory fits those

bits just as well.

The complement is !. “Not” flips every bit, and `from_bits` keeps the twelve that mean notes:

```
pub fn complement(self) -> PitchSet {  
    PitchSet::from_bits(!self.bits())  
}
```

Inverting turns each pitch class p into $12 - p$, which reflects the set around C:

```
pub fn invert(self) -> PitchSet {  
    self.iter()  
        .map(|pitch_class| PitchClass::from_semitone(12 -  
pitch_class.semitone()))  
        .collect()  
}
```

The most compact form is the smallest number. This is the neat part. Move a set down so that its first note is C, bit 0. Its last note is now its highest bit, so a smaller span means a smaller highest bit, which means a smaller number. On a tie, the next bit down is the note before last, which is exactly Rahn’s tie-break. Comparing two numbers *is* Rahn’s rule:

starting on	moved to C	bits	number
C	0 4 7 11	1000_1001_0001	2193
E	0 3 7 8	0001_1000_1001	393
G	0 4 5 9	0010_0011_0001	561
B	0 1 5 8	0001_0010_0011	291

So the normal form is “the start that gives the smallest number”, and the prime form “the smallest number of all 24 transpositions and inversions”:

```

pub fn normal_form(self) -> Vec<PitchClass> {
    let Some(start) = self
        .iter()
        .min_by_key(|start| self.relative_to(*start).bits())
    else {
        return Vec::new();
    };
    self.relative_to(start)
        .iter()
        .map(|pitch_class| pitch_class.transpose(start.semitone() as i32))
        .collect()
}

pub fn prime_form(self) -> PitchSet {
    let inverted = self.invert();
    (0..12)
        .flat_map(|n| [self.transpose(n), inverted.transpose(n)])
        .min_by_key(|set| set.bits())
        .unwrap_or(self)
}

```

`min_by_key` walks an iterator and keeps the item with the smallest key. In `prime_form`, `flat_map` turns each `n` into an array of two sets, so the iterator yields 24 sets, and the smallest is the prime form. No sorting and no lists of intervals: a few integer comparisons. `relative_to(start)` moves the set down until `start` is C.

How one set is moved onto another, as Tn or TnI , is a search over the same 24 moves:

```

pub fn moved_to(self, other: PitchSet) -> Option<(u8, bool)> {
    let inverted = self.invert();
    (0..12)
        .find(|&n| self.transpose(n) == other)
        .map(|n| (n, false))
        .or_else(|| {
            (0..12)
                .find(|&n| inverted.transpose(n) == other)
                .map(|n| (n, true))
        })
}

```

`or_else` runs its closure only when the first search found nothing, so a transposition is always preferred to an inversion. A class is **symmetric** when inverting its prime form gives a transposition of it, as for the diminished triad: upside down, it is the same set moved.

The interval vector is the one place that counts pairs the plain way, a loop inside a loop. With at most twelve notes that is at most 66 pairs:

```
pub fn interval_vector(self) -> [u8; 6] {
    let notes = self.semitones();
    let mut vector = [0; 6];
    for (i, low) in notes.iter().enumerate() {
        for high in &notes[i + 1..] {
            let interval = high - low;
            let class = interval.min(12 - interval);
            vector[class as usize - 1] += 1;
        }
    }
    vector
}
```

`interval.min(12 - interval)` folds an interval and its inversion together: 7, a fifth, becomes 5, a fourth.

224 classes in a table

The prime form can be computed, but the Forte *number* cannot: it is only a place in Forte's list. So `set_class.rs` holds the list, a constant array of 224 entries:

```
const SET_CLASSES: [SetClass; 224] = [
    SetClass::new(0, 1, false, 0),
    SetClass::new(1, 1, false, 0b00000000000001),
    SetClass::new(2, 1, false, 0b00000000000011),
    // ...
    SetClass::new(4, 20, false, 0b000100100011),
    // ...
    SetClass::new(12, 1, false, 0b111111111111),
];
```

`SetClass::new` and `PitchSet::from_bits` are `const fn`, so the whole table is built while compiling and stored with the program; looking a set up costs nothing at start-up. The `false` or `true` says whether the class has a Z in its name. `0b` writes a number in binary, so the prime form [0158] can be read off the bits.

A table of 224 lines of numbers is easy to get wrong, so the tests check it against the theory rather than trusting it:

- every entry is its own prime form, and the entries are in order of size and number;
- there are 1, 1, 6, 12, 29, 38, 50, 38, 29, 12, 6, 1 and 1 classes of 0 to 12 notes, the counts every textbook gives;
- **all 4,096 sets** are looked up, and each lands in a class of its own size;
- a class has a Z exactly when another class of its size has the same interval vector;

- for 5-20, where Forte and Rahn pack the notes differently, the prime form is Rahn's.

Because of the third test, the lookup can state what it expects instead of handling a “not found” that cannot happen:

```
pub fn of(set: PitchSet) -> &'static SetClass {
    let prime = set.prime_form();
    SET_CLASSES
        .iter()
        .find(|class| class.prime == prime)
        .expect("every prime form is in the table")
}
```

`expect` panics with that message if the `Option` is `None`. Using it is a promise, and the test is what keeps the promise. The function returns `&'static SetClass`: a borrow of an entry in the constant table, which lives as long as the program, so a caller can keep it as long as it likes.

The class writes its own names. `vector_text` writes the six counts as digits, `<001110>`, the way the books do, until one of them reaches 10: all twelve notes hold twelve of every interval but the tritone, and that is written with spaces, `<12 12 12 12 12 6>`, so it cannot be misread. Some classes also have a familiar name, from a short list of examples: “major or minor triad”, “whole-tone scale”, “octatonic (diminished) scale”, “Petrushka chord”.

The set browser

Sets..., in the piano roll's toolbar, opens the **Pitch-class sets** window: every class of the size picked (2 to 10 notes), each with its Forte number, prime form, interval vector and familiar name, such as “3-11 [037] <001110> major or minor triad”. The search box finds a class by number or by name (“4-Z15”, “triad”, “octatonic”), and with two or more notes selected in the piano roll, **Show it** jumps to their class. Picking a class shows its Z-partner if it has one, the class of its complement, and whether it is symmetric. **Move it up** and **Inverted (I)** choose one member of the class: T3 is the set three semitones up, T3I its inversion moved up three. Then:

- ► **Hear it** plays the set on the selected instrument;
- **Draw with it** makes the set the piano roll's chord stamp, so each click places the whole set above the note clicked (`ChordStamp::Set`, next to the chord stamps of [Chords: building, naming and inversions](#));

- **Use as the key** makes the set the song's key: the piano roll highlights its notes and the harmony keys play chords stacked from it.

A set of three notes is also one of Kröpf's micromodes, and the window links to it (see [Micromodes](#)).

The piano roll names sets without the window, too. Under the toolbar, next to the chord at the step under the pointer, it shows that step's set when three or more notes sound there, and the set of the selected notes when two or more are selected:

Set: 3-11 • normal form G# B D# • prime form [037] • intervals <001110> • major or minor triad

A key as a set: the same notes, another name

Picking **pitch-class set** in a Key menu could mean two things: play in another scale, or name the same notes another way. Ferrous Lineup takes the second meaning, because a set is a notation. C major already *is* a pitch-class set, C D E F G A B, the class 7-35. So switching the key from C major to “pitch-class set” keeps every note of the song and only shows the same harmony in set notation:

Harmony key	Notes	Set class
C	C E G	3-11
Dm	D F A	3-11
Em	E G B	3-11
F	F A C	3-11
G	G B D	3-11
Am	A C E	3-11
Bdim	B D F	3-10

The table is a lesson on its own. Major and minor triads are the **same** class, 3-11, because a minor triad is a major triad upside down, and a set class counts a set and its inversion as one. The diminished triad is another class, 3-10.

Switching between scales is different: from major to minor, E, A and B become E $\flat$, A $\flat$ and B $\flat$, and the song moves (see [Scales, modes and keys](#)). A set you choose in the set browser with **Use as the key** may be any set at all.

`set_key`, in the app, picks the set the menu switches to:

```

fn set_key(ctx: &egui::Context, scale: Scale, root: PitchClass) -> Scale {
    let up = scale.root.semitones_up_to(root);
    if let Some(group) = scale
        .tone_row()
        .and_then(|row| row.segments(3).first().copied())
    {
        return Scale::set(root, group.transpose(up));
    }
    if scale.kind.is_tonal() && scale.kind != ScaleKind::Chromatic {
        let set = Scale::set(root, scale.pitch_set().transpose(up));
        // Gb major as a set still writes its notes Gb Ab Bb Cb...
        return if up == 0 {
            set.with_spelling(scale.spelling())
        } else {
            set
        };
    }
    let set = remembered_set(ctx);
    let start = set.normal_form().first().copied().unwrap_or(root);
    Scale::set(root, set.transpose(start.semitones_up_to(root)))
}

```

From a key it is the key's own scale; from a twelve-tone row, the row's first three notes, which are what the harmony keys were playing; from the chromatic scale, which holds every note and says nothing about the harmony, the set used last. A set whose notes are exactly a scale on its root is spelled as that scale, and moves on as that key when the key changes again (`set_as_scale` and `as_key` , in [Scales, modes and keys](#)).

One harmony, four names

The chord sheet applies the same rule to every harmony on it. Its top row has a **Show** menu: **As written**, **Chords**, **Pitch-class sets** or **Micromodes**. The notes stay the same, only their names change, and nothing in the song is touched:

Written	Chords	Pitch-class sets	Micromodes
C	C	3-11 T7I	Acorde menor-mayor T7I
Dm	Dm	3-11 T2	Acorde menor-mayor T2
G	G	3-11 T2I	Acorde menor-mayor T2I
Am	Am	3-11 T9	Acorde menor-mayor T9
Bdim	Bdim	3-10 T11	Acorde disminuido T11
G7	G7	4-27 T7I	4-27 T7I
3-11 T2	Dm	3-11 T2	Acorde menor-mayor T2

Written	Chords	Pitch-class sets	Micromodes
Menor 3	Menor 3	3-3	Menor 3

How to read it:

- **Tn and Tnl.** Every class has a prime form that starts on C. For 3-11 it is [0, 3, 7], C E♭ G, a minor triad. **T9** means “the prime form moved up 9 semitones”: A C E, A minor. **Tnl** means “turned upside down first, then moved up *n*”. Upside down, [0, 3, 7] is 0, -3 and -7: C, A and F. Moved up 7, that is G, E and C, C major. So every major chord has an l in its name, and no minor chord does. Dm (T2) and G (T2l) share the number 2 but not the notes. The prime form itself, T0, is written with no T at all: Menor 3 is C D♭ E, 3-3 on C.
- **A micromode is always three notes.** Kröpfl’s twelve micromodes are the twelve classes of three notes, 3-1 to 3-12, with the Spanish names he gave them. The micromode 3-11 is **Acorde menor-mayor**, the minor-major chord, since the major and minor triads are one class. G7 has four notes, so it is no micromode, and the Micromodes view writes its set.
- **Chords** turns a set back into a chord **when its notes make one**: the set 3-11 T2 is D F A, so the view writes Dm, with the root spelled as the key spells it. A set that makes no chord the app knows, such as 3-1 or Menor 3, keeps its name. **N.C.** (no chord) stays N.C. in every view, and a row form (P0, RI7) stays a row form: a row is a line of twelve notes, not one group.

A polychord is two groups of notes, one over the other, and merging them into one set would lose what makes it a polychord. So each chord is named on its own, written upper first as the polychord is:

E♭7|F♭ → 4-27 T3I|3-11 T11I

In Rust: one match decides

`Notation` is an enum of four variants, `#[default]` on `Written` so the sheet starts as it always was. Renaming a harmony is one `match` on a **tuple** of the notation and the harmony. Rust tries the arms from the top, so their order is the rule: first what never changes, then the Chords view, then two chords at once, then everything else.

```

pub fn rename(self, harmony: HarmonyName, key: Scale) -> Vec<HarmonyName>
{
    let group = |set: PitchSet| self.name_group(set, key);
    match (self, harmony) {
        (Notation::Written, _)
        | (_, HarmonyName::NoChord)
        | (_, HarmonyName::Row(_))
        | (Notation::Chords, HarmonyName::Chord(_) |
HarmonyName::Polychord(..)) => {
            vec![harmony]
        }
        (Notation::Chords, HarmonyName::Set(set) |
HarmonyName::Micromode(set)) => {
            match chord_of(set) {
                Some(chord) => vec![HarmonyName::Chord(ChordSymbol::new(
                    key.spell_root(chord),
                    chord.quality,
                ))],
                None => vec![harmony],
            }
        }
        (_, HarmonyName::Polychord(upper, lower)) => {
            // The lower chord with its bass, then the upper chord.
            let lower = notes_of(lower);
            vec![group(lower), group(upper.chord().pitch_set())]
        }
        (_, HarmonyName::Chord(symbol)) => vec![group(notes_of(symbol))],
        (_, HarmonyName::Set(set) | HarmonyName::Micromode(set)) => vec!
[group(set)],
    }
}

```

The first arm uses **or-patterns** at two levels: `A | B | C` between whole tuples, and `Chord(_) | Polychord(..)` inside one. An or-pattern that binds a name, as `Set(set) | Micromode(set)` does, must bind it in every alternative, with the same type. The function returns a `Vec` because a polychord becomes two names and everything else one.

`chord_of` finds the chord a set makes. It goes through the chord qualities from the simplest, the major triad, to the most complex, and for each one through the notes of the set, in normal form, as possible roots:

```
pub fn chord_of(set: PitchSet) -> Option<Chord> {
    let roots = set.normal_form();
    ChordQuality::ALL.into_iter().find_map(|quality| {
        roots
            .iter()
            .find(|&&root| ChordQuality::identify(set.relative_to(root))
== Some(quality))
            .map(|&root| Chord::new(root, quality))
    })
}
```

`identify` says which quality the intervals above a root are, if any. Because the qualities are the outer loop, the quality listed first wins whatever its root: C E G A is both Am7 and C6, and is found as Am7, since seventh chords come before sixth chords in the list. `&&root` in the closure's pattern takes apart a reference to a reference: `iter` gives `&PitchClass`, and `find` passes its closure a reference to that.

Reading the notes themselves

Renaming the sheet's harmonies is half of what an analyst does with sets. Music without chord symbols, a Webern concerto or a page of the Rite of Spring, is cut into sets **from its notes**, in two ways:

- **vertically**, the notes that sound together, as a stacked chord or a cluster is heard;
- **horizontally**, the notes one after another in a line, as a twelve-tone row is cut into groups of three or six.

With **Pitch-class sets** or **Micromodes** shown, a second menu picks which: **The sheet's harmonies** (renamed, as above), **Notes together (vertical)** or **Notes in a line (serial)**. Reading notes, the sheet shows the groups it finds, each where it starts, and leaves what is written alone: typing in a bar still writes its harmony, and **As written** shows it again. Two more choices say how many notes make a group (2 to 12, or **All notes** sounding together; a micromode always has three) and whose notes are read. **Auto** reads the tune for a line (the track called Lead, or else the highest one, as the lead sheet picks it), because a line through every instrument at once would mix the melody with the chords; for notes together it reads every track, since a chord is usually shared between instruments.

Notes together

Wherever a note starts, `together` takes all the notes held at that moment, from the lowest up, keeps one of each pitch class (an octave doubling is the same note in a set) and cuts them into groups of the size chosen. Two small rules keep the groups musical:

- a single note left over at the top joins the group below it: seven notes in threes are 3 + 4, not 3 + 3 + 1, since one note is no harmony;
- a moment where fewer than two different notes sound is skipped.

```
fn chunks(notes: Vec<Note>, size: Option<usize>) -> Vec<NoteGroup> {
    if notes.is_empty() {
        return Vec::new();
    }
    let Some(size) = size else {
        return vec![NoteGroup { notes }];
    };
    let mut groups: Vec<NoteGroup> = notes
        .chunks(size.max(1))
        .map(|chunk| NoteGroup {
            notes: chunk.to_vec(),
        })
        .collect();
    if groups.len() > 1 && groups.last().is_some_and(|last|
last.notes.len() == 1) {
        let left = groups.pop().map(|last|
last.notes).unwrap_or_default();
        if let Some(under) = groups.last_mut() {
            under.notes.extend(left);
        }
    }
    groups
}
```

`chunks` on a slice cuts it into pieces of `size`, the last one maybe shorter. `pop` takes the last group out, and `last_mut` reaches the one that is now last, to add the leftover note to it.

The Augurs chord shows why the first rule matters. From the bottom up the strings play F_b A_b C_b, then G B_b D_b, then E_b on top. In threes that is F_b A_b C_b | G B_b D_b | E_b, and the lone E_b joins the group below: G B_b D_b E_b, which is E_b7. The sheet writes **4-27 T3|3-11 T11|**, the two chords Stravinsky put together, found from the notes alone. With **All notes**, the seven notes are one set, **7-32 T7**: the class of the harmonic minor scale, and the notes are exactly A_b harmonic minor. Two ways to look at one chord: two familiar chords stacked, or one scale sounding at once.

The Petrushka chord works the same way. C E G under F# A# C#, in threes, is **3-11 T1I|3-11 T7I**, F# major over C major, and all six notes are **6-30 T1I**, the class often called the Petrushka chord itself.

Notes in a line

`in_line` takes the notes one after another, in the order they start (notes struck together from the lowest up), and every so many notes make a group. Two rules matter here:

- a note struck again straight after itself, at the same pitch, counts once: it is the same note played again;
- any other repeat counts: the same note in another octave, or coming back later, is a new note of the line.

The row of Webern's Concerto, Op. 24, is the classic example:

```
B B $\flat$  D | E $\flat$  G F $\sharp$  | A $\flat$  E F | C C $\sharp$  A
```

In threes, every group is **Menor 3** (3-3, [014]): Webern built the whole row from one three-note cell, moved and turned upside down. In sixes, both halves are **6-20**, the hexatonic set. All twelve in one group are the aggregate, 12-1.

A group that is the one before it again is written once, where it starts, as the sheet writes a chord only where it changes (`merge_repeats`).

Reading only when something changes

Cutting a whole song into groups and naming them takes milliseconds for a long song: too slow to repeat sixty times a second while it plays. So the view keeps what it read, and what it read it from:

```
struct Read {  
    from: (Notation, Grouping, Option<usize>, Option<usize>),  
    notes: Vec<HeardNote>,  
    keys: KeyMap,  
    shown: Rc<[Shown]>,  
}
```

Each frame it gathers the song's notes, which is quick, and compares them, the choices and the keys with what it read last time. Only when something differs, a new

choice in a menu, a note edited in the piano roll, a key change, does it cut and name the groups again.

The result is an `Rc<[Shown]>`, a reference-counted pointer to a slice. `Rc` is the single-threaded cousin of `Arc` (see [Instruments: drums, sampler and Lineup Osc](#)): cloning it does not copy the names, it only adds one to a counter. The chord sheet clones it every frame to draw the bars while the cache keeps its own, and the names stay in one place in memory. The whole user interface runs on one thread, so `Rc` is enough; `Arc` would work too but would pay for thread safety that is not needed.

The groups are sorted by the step they start on. To draw one bar, the sheet needs the groups that start inside it, and `partition_point` finds them by **binary search**:

```
let first = heard.partition_point(|shown| shown.start < bar.start);
let end = heard.partition_point(|shown| shown.start < bar.end());
let shown = heard[first..end].to_vec();
```

On a sorted slice, “starts before the bar” is `true` for a first part and `false` for the rest, and `partition_point` returns where the `false` part begins. It looks at about $\log_2(n)$ items instead of all of them: about 11 for a song with 2,000 groups.

Micromode names are long, and “Acorde menor-mayor T9” does not fit in a quarter of a bar. Where the whole name does not fit, a set or a micromode is written by its class alone, “Acorde menor-mayor” or “3-11”, and past that as “...”; the hint under the pointer always gives the whole name and its notes. How the sheet fits names into bars is in [The chord sheet: model, layout and text](#).

The menus’ choices are a small struct kept in egui’s memory, which the app saves to disk, so they are there the next time it starts:

```
#[derive(Debug, Clone, Copy, PartialEq, Serialize, Deserialize)]
#[serde(default)]
struct Choices {
    notation: Notation,
    grouping: Grouping,
    /// How many different notes a group of notes sounding together
    /// holds, or `None` for all of them.
    together: Option<usize>,
    /// How many notes a group of a line holds.
    line: usize,
}
```

`#[serde(default)]` on the struct means a field missing from what was saved takes its value from `Default`. If a later version adds a field, the choices saved by this one still load.

Testing it

`set_class.rs` tests the theory itself: every set belongs to a class of its size, the table holds prime forms in order with the textbook counts, well-known classes have their well-known names (3-11, 4-28, 4-Z15, 7-35, 6-35, 5-35), Rahn and Forte differ for 5-20, Z-related classes share their intervals, normal forms pack the notes tightly, and the complement of the major scale is the pentatonic scale.

`chord_sheet/notation.rs` tests the views: the same notes in every notation, and every renamed harmony with exactly the notes of the written one; sets that make a chord written as the chord (3-11 T2 is Dm, C E G A is Am7, the fourths 4-23 T2 are A7sus4); two chords at once as two sets; the Petrushka and Augurs chords read vertically; Webern's row read in a line; and every note of a line counting but a repeated one:

```
#[test]
fn sets_that_make_a_chord_are_written_as_the_chord() {
    assert_eq!(renamed(Notation::Chords, "3-11 T2"), "Dm");
    assert_eq!(renamed(Notation::Chords, "{C E G}"), "C");
    assert_eq!(renamed(Notation::Chords, "Acorde disminuido T11"),
"Bdim");
    // C E G A is both C6 and Am7: the simpler, more common chord.
    assert_eq!(renamed(Notation::Chords, "{C E G A}"), "Am7");
    // ...
}
```

A test of the window (`ux_tests.rs`) plays the Four chords style, clicks through the Show menu, checks the names, then switches back and checks that the project is exactly as it was.

Exercises on sets are in [Exercises: music theory in code](#).

Twelve-tone rows and their rules

Files: `crates/lineup-core/src/harmony/row.rs`, `crates/lineup-core/src/serial.rs`, `crates/lineup-core/src/library/rows.rs`, `crates/lineup-core/src/harmony/voice_leading/serial.rs` and `crates/lineup-app/src/twelve_tone.rs`

Most of Ferrous Lineup assumes music in a key: a home note, chords that lean towards it, harmony keys numbered I to vii°. A large part of the music of the 20th century has no key at all, and much of it was written with **twelve-tone rows** (*dodecafonismo* in Spanish, also called *series*): all twelve notes in an order that the whole piece is made of. In the app, **twelve-tone row** is one of the kinds of key, **Row...** in the piano roll opens a window to type, draw or pick a row, see its **matrix** of 48 forms and write any form into the pattern, the Library has a group of row styles, and the **Voice leading** check knows the rules twelve-tone composers wrote by. This chapter covers all four, and the Rust behind them: a newtype that is always a real row, arithmetic on twelve numbers, backtracking, a builder that runs while compiling, bit sets and dynamic programming.

The music, briefly

In C major, C is home. G pulls towards it, B pulls hard, and a piece that ends on C sounds finished. Around 1921 Arnold Schoenberg looked for a way to write music in which no note is home. His answer was to use all twelve notes equally, in a fixed order that he chose for each piece: the **row**. No note comes back until the other eleven have sounded, so none of them gets to be the centre.

This is the row of Schoenberg's Suite for Piano, Op. 25, from 1925, one of the first pieces built entirely from one:

E F G C# F# D# G# D B C A A#

A row would be dull if it could only be played one way, so it comes in four:

form	how	Schoenberg's row on E
P, prime	as written	E F G C# F# D# G# D B C A A#

form	how	Schoenberg's row on E
I , inversion	every interval upside down: where P goes up a semitone, I goes down one	E D# C# G D F C F# A G# B A#
R , retrograde	P backwards	A# A C B D G# D# F# C# G F E
RI , retrograde inversion	I backwards	A# B G# A F# C F D G C# D# E

Each of the four can start on any of the twelve notes, which gives **48 forms**. They are named by a letter and a number: the number is the note the form starts on, with C as 0, C# as 1, up to B as 11. So the row above is **P4** (it starts on E), and **I0** is its inversion starting on C. A retrograde gets the number of the form it reverses: **R4** is P4 backwards, so it *ends* on E.

Some books number the forms from the row's own first note instead, so that the row as written is always P0. Then Schoenberg's row is P0, and what Ferrous Lineup calls I11 is I7 there. The forms are the same; only their numbers move. Ferrous Lineup numbers a form by the note it starts on, because then a label names a note you can find on the keyboard.

The matrix

All 48 forms fit in one 12×12 square. The row goes along the top, its inversion from the same note down the left side, and each line *i* is the row moved to start on the note at the left of that line:

	I4	I5	I7	I1	I6	I3	I8	I2	I11	I0	I9	I10	
P4	E	F	G	C#	F#	D#	G#	D	B	C	A	A#	R4
P3	D#	E	F#	C	F	D	G	C#	A#	B	G#	A	R3
P1	C#	D	E	A#	D#	C	F	B	G#	A	F#	G	R1
P7	G	G#	A#	E	A	F#	B	F	D	D#	C	C#	R7
...													
	RI4	RI5	RI7	RI1	...								

Read a line left to right and it is a P form; right to left, an R form. Read a column top to bottom and it is an I form; bottom to top, an RI form. The labels around the square say which: P on the left, R on the right, I along the top, RI along the bottom. In the Row window, click any label to choose that form, and its line or column lights up.

The diagonal from top left to bottom right is the same note all the way down (E, here): a quick check that a matrix was filled in correctly.

Groups of notes taken from a row, such as its four groups of three, are **pitch-class sets**, and the theory that names them has a chapter of its own: [Pitch-class sets, and one harmony with four names](#).

A row in Rust: `ToneRow`

```
[derive(Debug, Clone, Copy, PartialEq, Eq, Hash, Serialize, Deserialize)]
[serde(try_from = "String", into = "String")]
pub struct ToneRow([u8; 12]);
```

A row is twelve numbers, the semitones above C of its notes. That is a **newtype** around an array: the array is private to the module, so the only ways to make a `ToneRow` are functions that check it:

```
pub fn new(semitones: [u8; 12]) -> Option<Self> {
    let set: PitchSet = PitchSet::from_semitones(&semitones);
    (set.len() == 12 && semitones.iter().all(|&n| n <
12)).then_some(ToneRow(semitones))
}
```

Collect the twelve numbers into a set of pitch classes, and if the set has twelve members, no note was repeated. Every `ToneRow` in the program is therefore a real row: the **invariant** is checked once, here, and nothing else has to worry about it.

Files: `try_from` and `into`

The two `serde` attributes on the struct say: “to save a `ToneRow`, turn it into a `String`; to read one, read a `String` and try to turn it into a `ToneRow`”. The conversions are ordinary trait implementations:

```

impl TryFrom<String> for ToneRow {
    type Error = String;

    fn try_from(text: String) -> Result<Self, String> {
        ToneRow::parse(&text)
    }
}

impl From<ToneRow> for String {
    fn from(row: ToneRow) -> Self {
        let numbers: Vec<String> =
            row.0.iter().map(u8::to_string).collect();
        numbers.join(" ")
    }
}

```

So a project file holds "4 5 7 1 6 3 8 2 11 0 9 10", which a person can read, instead of a list of twelve numbers in brackets. Saving cannot fail, so it is a plain `From`; reading can, so it is a `TryFrom` with an error. Because reading goes through `parse`, a file with a note written twice is refused with a message, rather than producing a broken row. `PitchSet` has the same pair, so a set is saved as "0 3 6 9".

`parse` also reads the box at the top of the Row window. It takes note names in letters (E F G Db Gb Eb Ab D B C A Bb) or in solfège (Mi Fa Sol Reb Solb Mib Lab Re Si Do La Sib), numbers (4 5 7 1 ...) or the compact form twelve-tone books use, one character a note with **T** and **E** for ten and eleven (45716382E09T). When a note comes twice it says which, and which note is missing instead.

A name is read in letters first, and in solfège when that fails. No syllable is also a letter name, since none starts with A to G but `Do`, `Fa`, whose second letter is no sharp or flat, so the two never clash. Before reading solfège, `parse_name` writes the name with a capital and the rest in small letters, so `SOL#` and `sib` read as `Sol#` and `Sib`, where the small `b` is a flat:

```

fn parse_name(token: &str) -> Result<u8, String> {
    parse_letters(token).or_else(|error| {
        let mut chars = token.chars();
        let first = chars.next().map(|c| c.to_uppercase().collect::
<String>());
        let written = first.unwrap_or_default() +
            &chars.as_str().to_lowercase();
        NoteName::read_as(&written, NoteNaming::Solfège)
            .filter(|(_, rest)| rest.is_empty())
            .map(|(name, _)| name.pitch_class().semitone())
            .ok_or(error)
    })
}

```

`or_else` runs its closure only on an error, and `ok_or(error)` gives back the letters' error when solfège fails too, so a wrong name is explained in the terms most people type.

From the notes reads a row from the piano roll: the first twelve different pitch classes among the selected notes (or all of them), in the order they start.

`ToneRow::from_notes` skips a note whose pitch class is already in the row, so you can draw freely and repeat notes, as long as all twelve turn up. Both end in a third conversion, from a `Vec<u8>`:

```
impl TryFrom<Vec<u8>> for ToneRow {
    type Error = String;

    fn try_from(semitones: Vec<u8>) -> Result<Self, String> {
        let array: [u8; 12] = semitones
            .try_into()
            .map_err(|v: Vec<u8>| trf("a row has 12 notes, not {}", &
[&v.len()])))?;
        ToneRow::new(array).ok_or_else(|| tr("a row has each of the 12
notes once")).to_string()
    }
}
```

A `Vec` can hold any number of notes and an array exactly twelve, so the standard library's `try_into` from one to the other can fail. When it does, it hands the vector back unchanged as the error, and `map_err` turns that into a message that says how many notes there were. `ok_or_else` then turns `new`'s `None` into an error too. The closures run only when something is wrong, so a good row never builds a message.

Every form from one function

```
pub fn form(self, label: RowLabel) -> ToneRow {
    let first = self.0[0];
    let n = label.number;
    let mut notes = self.0.map(|note| {
        let up = (note + 12 - first) % 12;
        if label.form.is_inverted() {
            (n + 12 - up) % 12
        } else {
            (n + up) % 12
        }
    });
    if label.form.is_backwards() {
        notes.reverse();
    }
    ToneRow(notes)
}
```

Each note is first measured as `up`, its distance above the row's first note (0 to 11). A P form puts it the same distance above the new start `n`; an I form puts it the same distance *below*. That is all an inversion is. The `+ 12` before each `%` keeps the numbers from going below zero: they are `u8`, and `4 - 7` would not wrap around the octave but stop the program in a debug build.

`self.0.map(...)` is the `map` of an **array**, not of an iterator: it calls the closure on each of the twelve items and gives back a new `[u8; 12]`, with no `Vec` and no `collect`. The length is part of the type, so the result is still exactly twelve notes.

With `form`, the matrix is two lines:

```
pub fn matrix(self) -> [[u8; 12]; 12] {
    let inversion = self.form(RowLabel::new(RowForm::Inversion,
self.0[0]));
    inversion
        .0
        .map(|start| self.starting_on(PitchClass::from_semitone(start)).0)
}
```

The inversion from the row's first note is the left column; each of its notes starts one line.

Going the other way, “which form of this row is that?”, tries all 48 until one matches:

```
pub fn label_of(self, other: ToneRow) -> Option<RowLabel> {
    RowForm::ALL.into_iter().find_map(|form| {
        (0..12)
            .map(|n| RowLabel::new(form, n))
            .find(|&label| self.form(label) == other)
    })
}
```

`find_map` stops at the first form that gives `Some`, and `find` at the first number that matches, so the search ends as soon as the form is found.

What makes a row interesting

Under the matrix, the Row window lists what is special about the row. Each line is a short function.

The steps. `intervals()` gives the eleven steps from note to note, in semitones going up. For Schoenberg's row they are 1 2 6 5 9 5 6 9 1 9 1. It is built with `std::array::from_fn`, which makes an array by calling a closure with each index:

```
pub fn intervals(self) -> [u8; 11] {
    std::array::from_fn(|i| (self.0[i + 1] + 12 - self.0[i]) % 12)
}
```

The return type says eleven, so `from_fn` calls the closure eleven times, and `i + 1` never runs past the twelfth note.

All-interval rows. When the eleven steps are all different, every interval from a semitone to a major seventh appears exactly once. Alban Berg's Lyric Suite (1927) starts with such a row: F E C A G D G# C# D# F# A# B, with the tritone right in the middle. They are rare: of the 39,916,800 rows that start on C, only 3,856 are all-interval. So the 🎲 **All-interval** button cannot just shuffle notes until it finds one; it builds the row a note at a time:

```

fn all_interval_from(notes: &mut Vec<u8>, random: &mut Random) ->
Option<Vec<u8>> {
    if notes.len() == 12 {
        return Some(notes.clone());
    }
    let used: Vec<u8> = notes.windows(2).map(|w| (w[1] + 12 - w[0]) %
12).collect();
    let mut intervals: Vec<u8> = (1..12).filter(|i|
!used.contains(i)).collect();
    for i in (1..intervals.len()).rev() {
        intervals.swap(i, random.below(i + 1));
    }
    let last = *notes.last()?;
    for interval in intervals {
        let next = (last + interval) % 12;
        if notes.contains(&next) {
            continue;
        }
        notes.push(next);
        if let Some(row) = all_interval_from(notes, random) {
            return Some(row);
        }
        notes.pop();
    }
    None
}

```

This is **backtracking**: try each step not used yet, in a random order; if the note it lands on is free, add it and carry on from there (the function calls itself); if that leads nowhere, take the note back (`pop`) and try the next step. When no step works, return `None` so the caller takes back *its* note. `windows(2)` hands out every pair of neighbours, `[a, b]` , which is exactly what a step is. All the calls share one vector through `&mut` , so trying a note costs a `push` and a `pop` , not a copy.

Random rows. The 🎲 **Random** button shuffles the twelve notes with the **Fisher-Yates shuffle**, the same loop that shuffles the intervals above: walk from the last place to the first, and swap each with a place chosen at random among those not done yet. Every order is then equally likely. The random numbers come from a tiny generator, *SplitMix64*, written out in a few lines instead of adding a library:

```

pub(crate) fn next(&mut self) -> u64 {
    self.0 = self.0.wrapping_add(0x9E37_79B9_7F4A_7C15);
    let mut z = self.0;
    z = (z ^ (z >> 30)).wrapping_mul(0xBF58_476D_1CE4_E5B9);
    z = (z ^ (z >> 27)).wrapping_mul(0x94D0_49BB_1331_11EB);
    z ^ (z >> 31)
}

```

`wrapping_add` and `wrapping_mul` say “overflowing is fine here, just wrap around”: a plain `+` that overflows would stop a debug build. The same seed always gives the same row, which makes the tests repeatable. `pub(crate)` lets other modules of `lineup-core` use the generator (ear training picks its questions with it) while keeping it out of the crate’s public API.

Groups of notes. Composers often play a row in groups: four trichords (three notes each), three tetrachords, two hexachords. `segments(size)` cuts the row with `chunks_exact`, which hands out slices of exactly `size` notes and leaves out a shorter last one, and the window names each group’s set class (see [Pitch-class sets](#)). The tetrachords of Schoenberg’s row are three different classes, 4-12, 4-Z15 and 4-1; the trichords of the Four triads row are all 3-11, the major and minor triads.

Combinatoriality. Take the first six notes of a row. Some other form of the same row may start with exactly the *other* six. Play the two forms at the same time, and each half of the music holds all twelve notes, without one coming back early. Schoenberg built his later works on this, and it is called **hexachordal combinatoriality**:

```
pub fn combinatorial_forms(self) -> Vec<RowLabel> {
    let missing = PitchSet::from_semitones(&self.0[..6]).complement();
    [
        RowForm::Prime,
        RowForm::Inversion,
        RowForm::RetrogradeInversion,
    ]
    .into_iter()
    .flat_map(|form| (0..12).map(move |n| RowLabel::new(form, n)))
    .filter(|&label| PitchSet::from_semitones(&self.form(label).0[..6]) ==
missing)
    .collect()
}
```

The retrograde is left out because it always works (its first half is the row’s second half), so it says nothing about the row. Schoenberg’s Op. 25 row is combinatorial with I11: its first six notes, E F G C# F# D#, and the first six of I11, B A# G# D A C, make all twelve. The Lineup row, written for Ferrous Lineup, is combinatorial with I3.

The `move` in `move |n|` is needed because the inner closure is handed out of the outer one by `flat_map`, and it uses `form: move` makes it take its own copy of `form` instead of borrowing a variable that is about to go away.

The rows to start from

The **Famous and example rows** menu holds eight rows. Four come from famous works, and four were written for Ferrous Lineup to show what a row can do.

`ROW_PRESETS` is a `const` array of them, each with a name, where it comes from and a sentence about it.

row	from	what it shows
E F G C# F# D# G# D B C A A#	Schoenberg, Suite for Piano, Op. 25 (1925)	one of the first twelve-tone pieces; its last four notes, B C A A#, spell BACH backwards in German note names, where B is B $\flat$ and H is B
F E C A G D G# C# D# F# A# B	Berg, Lyric Suite (1927)	an all-interval row, with the tritone in the middle
A F# G G# E F B A# D C# C D#	Webern, Symphony, Op. 21 (1929)	a palindrome: backwards, it is the same row a tritone higher (R9 is P3)
B A# D D# G F# G# E F C C# A	Webern, Concerto, Op. 24 (1934)	four groups of three notes, all the same micromode, Menor 3
C D F# G# F E A C# B D# G A#	Lineup row	hexachordally combinatorial with I3
C E G D F# A A# C# F B D# G#	Four triads	four triads that still use every note once
C E D# G G# B F D C# A A# F#	Mirror row	a palindrome, like Webern's
C D F E B A D# G G# C# A# F#	All the intervals	an all-interval row

The works from 1925 to 1929 are in the public domain in the United States (95 years after publication), and their composers died in 1951, 1935 and 1945, so they are in the public domain where copyright lasts 70 years after the author's death too. Webern's Concerto was published in 1948, so it is in the public domain where copyright lasts 70 years after the author's death. Only the rows are used: twelve notes each.

Some rows do not have 48 *different* forms. Since Webern's R9 is its P3, every R form of that row is also a P form, and every RI form an I form: only 24 are different. Berg's row and the Mirror row are the same: read backwards, each is itself a tritone higher.

`label_of` tries the P forms first, so such a form is named as the P form it also is.

From a row to notes: serialism

A row says which note comes next, and nothing else. To make music of it, someone still has to decide how long each note is, how loud, in which octave, and whether some notes sound together. `RowWriting` holds those choices, and the controls under the matrix set them:

```
pub struct RowWriting {
    /// The form to start with, such as `P0`.
    pub label: RowLabel,
    pub chain: Chain,
    /// How many notes of the row sound together: 1 for a melody, 3 to
    /// play the row as four three-note chords (trichords), 6 for two
    /// hexachords.
    pub together: usize,
    /// How long each note (or chord) lasts, in steps, when the durations
    /// do not follow the row.
    pub step: usize,
    pub register: Register,
    /// The lowest note the row is written from.
    pub low: Note,
    /// Whether the durations follow the row (see the module
documentation).
    pub durations: bool,
    /// Whether the velocities follow the row's inversion.
    pub dynamics: bool,
    /// Whether the octaves follow the row's retrograde.
    pub octaves: bool,
}
```

Then (`chain`) says what comes after the form until the pattern is full: nothing (**Once**), the same form again (**Repeat**), or the four forms in turn with the same number (**P**, **R**, **I**, **RI**). **Together** is `together`, and **Each** is `step`. The three **registers** (the **Octaves** menu) decide the octaves when the row does not:

- **Close** puts every note in the octave above the lowest note;
- **Smooth** puts each note as near as it can be to the one before, which makes a singable line (a tritone, exactly halfway, goes up);
- **Wide** sends the notes low, high, middle, low, high, middle over three octaves, the leaping lines of Anton Webern.

Integral serialism

Around 1950, Olivier Messiaen, Pierre Boulez and Milton Babbitt took the idea further: if the pitches follow a series, why not the durations, the dynamics and the octaves too? This is **integral** (or total) serialism. Real pieces map series to durations and dynamics in many ways; Ferrous Lineup uses three simple ones, each reading a different form of the row so that the series do not move in step:

- **Durations:** the row's numbers (each note's distance above the first note, 0 to 11) become lengths. The k -th note lasts as many steps as the k -th number, plus one, so from 1 to 12 sixteenth notes.
- **Dynamics:** the numbers of the inversion become twelve velocities, from very soft to very loud: `28 + 9 * level`, from 28 to 127.
- **Octaves:** the numbers of the retrograde, counted in threes (`% 3`), pick one of three octaves.

For Schoenberg's row the numbers are 0 1 3 9 2 11 4 10 7 8 5 6, so with **Durations** ticked its notes last 1, 2, 4, 10, 3, 12... steps. Inside `write_row`, each note asks the three questions in turn:

```

let lasts = if writing.durations {
    usize::from(numbers[place % 12]) + 1
} else {
    step
};
// ...
let octave = if writing.octaves {
    // The retrograde's numbers, in threes.
    Some(numbers[11 - k % 12] % 3)
} else if writing.register == Register::Wide {
    // Low, high, middle, and round again, against the row.
    Some([0, 2, 1][k % 3])
} else {
    None
};
// ...
let velocity = if writing.dynamics {
    // The inversion's numbers.
    dynamic((12 - numbers[k % 12]) % 12)
} else {
    crate::DEFAULT_VELOCITY
};

```

`place` counts notes across forms, so with **P, R, I, RI** the series of durations carries on from one form into the next instead of starting again. `k` is `place` plus the note's place in its chord, so each note of a chord gets its own octave and loudness.

`write_row` is a pure function: a row, the choices and a pattern length in; a list of `NoteEvent`s out. The window calls it every frame to warn when the pattern is too short for the whole row ("The pattern only has room for 9 of the 12 notes"), and **Write P0 into current pattern** sends the same notes to the piano roll, which replaces the channel's notes through the usual edit, so **Ctrl+Z** brings the old ones back. **Save to new pattern** puts them in a new pattern instead.

A row as the key

twelve-tone row in the key menus, or **Use as the key** in the Row window for the form you chose, makes the row the key. In the code it is one more kind of key, `ScaleKind::Row(ToneRow)`, which carries its row with it ([Scales, modes and keys](#) shows how the rest of the code treats it). The row is stored starting on C, and the key's root says where it really starts, so P4 of Schoenberg's row is saved like this:

```

scale: (root: E, kind: Row("0 1 3 9 2 11 4 10 7 8 5 6")),

```

With a row as the key, the piano roll numbers its notes **1 to 12** on the keys, in the row's order, so you can draw the row by following the numbers. The harmony keys play the row's groups instead of chords, the way twelve-tone composers build their chords: **3 notes** gives four keys for the trichords, **1-3, 4-6, 7-9** and **10-12**; **4 notes** gives three keys, and **6 notes** two. They come from cutting the row in order:

```
ScaleKind::Row(_) => {
  let group = row_group(size);
  Some(
    notes
      .chunks(group)
      .enumerate()
      .map(|(i, chunk)| {
        let label = format!("{}", i * group + 1, i * group +
chunk.len());
        GroupKey::new(scale, Some(label), chunk.to_vec())
      })
      .collect(),
  )
}
```

Changing the key from C major to a row moves no notes, since there is no sensible place for E to go in a row; changing to the same row on another note moves them all, which is transposing. Switching from a row to **pitch-class set** keeps the row's first three notes as the set, which are what the harmony keys were playing (see [A key as a set: the same notes, another name](#)).

Row styles and demos

The Library's **Twelve-tone rows** group has seven styles:

style	what it does
Row melody	the Lineup row as a tune in eighth notes, through P, R, I and RI
Triad chorale	the Four triads row as chords, over its retrograde in the bass: twelve-tone music that sounds almost tonal
Hexachord pads	half the row as a chord, the other six notes from I3 as a melody: every bar has all twelve notes
Pointillism	single notes leaping over three octaves, durations and dynamics from the row, as in Webern
Total serialism	pitches, durations, dynamics and octaves all from the all-interval row, over its retrograde inversion in two-note

style	what it does
	chords
Twelve-tone lo-fi	the row's trichords over a lo-fi beat, its prime form in the bass: a row can groove too
Webern's trichords	the row of Webern's Concerto, its melody leaping over three octaves over its groups of three as chords

A harmony style is a chord progression and a way of playing it. A row style has no progression, so it is a row and a list of **voices**, each saying how one instrument plays the row. As with the rest of the Library, it is data:

```
RowStyle {
  name: "Triad chorale",
  // ...
  row: "Four triads",
  tempo: 72.0,
  time: TimeSignature::COMMON,
  bars: 4,
  voices: &[
    voice(KEYS, RowForm::Prime)
      .chain(Chain::FourForms)
      .chords(3, 4)
      .from(Register::Nearest, 55),
    voice(BASS, RowForm::Retrograde)
      .step(8)
      .from(Register::Close, 36),
  ],
  rhythm: "",
},
```

`voice` makes a plain voice (a melody in eighth notes from middle C), and each method changes one thing and returns the new voice:

```
impl RowVoice {
  const fn offset(self, offset: u8) -> Self {
    RowVoice { offset, ..self }
  }
  const fn chain(self, chain: Chain) -> Self {
    RowVoice { chain, ..self }
  }
  // ...
}
```

Taking `self` by value and giving back a changed copy with **struct update syntax** (`..self`) lets the calls chain, like a builder. And because they are `const fn`, the whole table can be a `static: pub static ROW_STYLES: [RowStyle; 7]` is built while compiling, with no code running at start-up.

Form arithmetic

The demos in **File > Demos > Twelve-tone** play each style four times, as the sections of a song: **P: the row, R: backwards, I: upside down** and **RI: both**, then P once more to end. With a rhythm, the drums come in after the first section, with a fill before the last. But a style's voices already use different forms: the Triad chorale's bass plays R. What does the bass play in the R section? R of R, which is the row forwards again: P.

The four forms behave like two switches, “upside down” and “backwards”, and combining two forms flips each switch once for each form that has it on:

```
pub fn combine(section: RowForm, voice: RowForm) -> RowForm {
  let flags = |form| match form {
    RowForm::Prime => (false, false),
    RowForm::Inversion => (true, false),
    RowForm::Retrograde => (false, true),
    RowForm::RetrogradeInversion => (true, true),
  };
  let (a, b) = (flags(section), flags(voice));
  match (a.0 != b.0, a.1 != b.1) {
    (false, false) => RowForm::Prime,
    (true, false) => RowForm::Inversion,
    (false, true) => RowForm::Retrograde,
    (true, true) => RowForm::RetrogradeInversion,
  }
}
```

`a.0 != b.0` on two `bool`s is an *exclusive or*: true when exactly one is true.

Mathematicians call this little system the **Klein four-group**: every form undoes itself (I of I is P), and the order does not matter. One more detail: the Hexachord pads melody plays I3, three semitones *above* the row's first note. Upside down, “three above” becomes “three below”, so in the I and RI sections that offset becomes $12 - 3 = 9$:

```
let offset = if inverted {
  (12 - self.offset % 12) % 12
} else {
  self.offset % 12
};
```

Playing a whole song as a row

As with the other styles, a row style can be tried two ways: stopped, it is written into a pattern of its own; while the song plays, it takes over the whole song. `use_row_style` goes through every pattern that has chords, a bass line or a melody, and writes the style there, reading the patterns in turn as P, R, I and RI so the song does not repeat

itself. The key becomes the row, starting on the song's key note, and endings stay as they are.

One rule decides what happens to the song's own parts: **the style takes over the harmony**. Tonal chords or a tonal bass under a row only clash, so the chords and bass channels the style does not use go quiet in that pattern, and the style's chords go on the channel that was playing the song's chords. The melody stays, unless the style has one of its own:

```
if chords.is_some() || plays(project, bass) || melody {
    let before = project.pattern(pattern).cloned();
    let quiet: Vec<usize> = [(KEYS, chords), (BASS, bass)]
        .into_iter()
        .filter(|(name, _)| !uses(name))
        .filter_map(|(_, channel)| channel)
        .collect();
    // ... empty the `quiet` channels, write the voices ...
    if project.pattern(pattern) != before.as_ref() {
        written += 1;
        change.harmony += 1;
    }
}
```

To know whether a pattern really changed, it keeps a copy from before (`cloned()` turns the `Option<&Pattern>` into an `Option<Pattern>`) and compares at the end. `as_ref()` turns the copy back into an `Option<&Pattern>` , so the two sides of `!=` have the same type. This function, `take_over_harmony` , takes the writing as a closure, so the [micromode styles](#) use it too.

A window that opens in front

egui puts a window that opens on top of the others. But the window whose **Row...** button opened it was clicked in the same frame, and a click raises the window it lands in, so both want the top, and the new one can stay behind. `Front` keeps a new window in front for its first few frames:


```

#[derive(Default)]
pub(crate) struct Front {
    /// The frame the window was last drawn in.
    drawn_in: u64,
    /// The frame it opened in.
    opened_in: u64,
}

impl Front {
    /// Called each frame the window `id` is drawn, before it is.
    pub(crate) fn keep(&mut self, ctx: &egui::Context, id: egui::Id) {
        let frame = ctx.cumulative_pass_nr();
        if self.drawn_in + 1 < frame {
            self.opened_in = frame;
        }
        self.drawn_in = frame;
        if frame - self.opened_in < 3 {
            ctx.move_to_top(egui::LayerId::new(egui::Order::Middle, id));
            ctx.request_repaint();
        }
    }
}

```

`cumulative_pass_nr` counts the frames egui has drawn. A window that was not drawn last frame has just opened, so it remembers that frame; for its first three frames it moves its own layer to the top and asks for another frame, so the three come at once instead of waiting for the mouse to move. To find its layer, the window needs its id, so the Row, Sets and Micromodes windows each set one with `.id(...)`, made with `egui::Id::unique` (*unique* because it names one thing in the whole program, as a window id must). The playlist does the same when a double-click opens a clip in the piano roll ([Double-click a clip to edit it](#)).

The Row window is as wide as the matrix and scrolls when the screen is short, so its buttons at the bottom stay in reach: `.resizable([false, true])` lets it get taller or shorter but not wider, and `.constrain_to(area)` keeps it inside the main window.

The rules of twelve-tone writing

The [voice-leading check](#) marks what classical part writing avoids: parallel fifths, a leading tone that does not rise, a seventh that does not fall. Every one of those rules is about a key. In music written with a row there is no tonic, so they mean nothing there. Twelve-tone music has rules of its own. Schoenberg wrote by them, and so did his students Berg and Webern; composition courses still teach them. So when the key is a **twelve-tone row**, the **Voice leading** tick box checks those rules instead.

What you see

- **A Row line** under the chord names. It names each form of the row that the check finds in the notes, such as **P4** or **P10**, over the notes that play it. Point at it to read the form's notes: "P4: the row starting on E: E F G C# F# Eb Ab D B C A Bb."
- **The Voices line** under it marks what twelve-tone writing avoids: **order** for a note out of the row's order, **rep** for a note repeated too early, **gap** for a form left unfinished, **8ve** for a note doubled in octaves, **tonal** for a chord that sounds like tonal harmony.
- **Outlines** around the notes concerned. Two doubled notes are joined by a line, and so are the notes of a tonal chord. A dashed outline shows the note the row wanted instead.
- **The summary line** above the chord rows sums it all up: "Twelve-tone rules, P4, P10, through all the tracks together: out of row order (2), octave doubling (1), sounds tonal (1)." Point at a step with a finding and the line says what happens there; hover over the line to read why.

With a **pitch-class set** as the key, nothing is checked. A set is a collection of notes with no order and no rules of its own, so there is nothing to check against, and the line under the chords says so.

The five rules

The row sets the order. A piece moves through forms of the row, each played note after note, in its order, so that everything you hear comes from the row. Notes next to each other in the row may sound together as a chord, in any order from the bottom up, and a note may be repeated straight away, or trilled with the note before it: that is still the same note of the row. Two mistakes are counted with care. If you **swap** two notes, or **leave one out**, that is one note out of place, not every note after it: with B played before D in P4, the check marks the B ("Melody plays B4 where P4 wants D, its note 8") and finds the rest of the row in order again.

No note comes back early. Before a note comes back, the other eleven sound. The row was made for that: no note is heard more often than the others, so none of them becomes a tonic. A note that comes back after other notes, before the form has sounded all twelve, is marked **rep**.

A form is played whole. When a form stops short and the next one starts, the notes it left out are missing from the twelve. The check marks the last note of such a form with **gap** and names what never sounded: "Row left unfinished: P4 stops after 8 of its 12 notes, with Melody's D5; B, C, A and Bb never sound before the next form starts."

No octave doublings. Doubling a note in octaves gives it weight. As Schoenberg explained, a note with weight can be heard as a root, or even as a tonic. The check marks two parts that play the same note together, and also a part that plays a note while another part still holds it. A part that doubles another all the way through, a line in octaves, is told once, where it starts.

No chords of tonal harmony. A major or minor triad, or a dominant seventh, reminds the ear of tonal music, and the ear then expects the music to go on as tonal music would. Schoenberg and Webern avoided them. The check marks a moment where everything sounding together makes one of these chords, and names it. **Berg** used them on purpose: the row of his Violin Concerto (1935) is built from triads, G minor, D major, A minor and E major. So read **tonal** as “this sounds like a key”, not “this is wrong”.

Reading the row as an analyst would

Nothing in the notes says which form of the row is playing, or where one form ends and the next begins. A student analysing a score has to find that out, and so does the check. It cuts the notes into **groups**, notes that start together, then tries every way of sharing the groups out among forms of the row, and keeps the way with the fewest problems: each note out of order, each note repeated early and each note a form leaves out counts one. A few things make the reading closer to what composers actually do:

- **Forms can overlap at the join.** A form may begin on the note, or the chord, that the form before ended on, so the two share it: an **elision**. If P4 ends with the chord A B $\flat$ C and P10 starts on B $\flat$, P10 takes the B $\flat$ from that chord and goes on with B.
- **A chord may share notes with the chord before.** Those notes count for the new form if it needs them, or as repeated notes if it does not.
- **One row or one per track.** All the tracks can share one form, as a tune and the chords under it often do, or each track can play forms of its own, as in a canon. The check reads the row both ways, keeps the reading with fewer problems, and the summary line says which one it kept.
- **Patterns join up.** Each pattern is checked on its own, but a row may run on from one pattern into the next. So the first form may have begun in the pattern before, as long as its notes here are in order: the Row line then says “...P4”, and the summary “P4 from note 3”. The last form may go on in the next pattern, so it is never marked unfinished.
- **Ties.** When two readings have equally few problems, the check prefers forms that start on their first note. If two forms still tie, it names the first in the order P, I, R, RI. Chords can make two forms impossible to tell apart: the row of Webern’s

Concerto is four groups of three notes, and its forms R0 and I1 have the same three notes in each group, in another order. In the Webern's trichords style the melody and the chords both play R0 after P0. The melody's notes come one at a time, so they can only be R0; the chords sound each group at once, so they fit R0 and I1 equally, and the check names them I1, which comes first. Either name is right.

The check follows the classic rules of Schoenberg's method. It does not judge **free atonal music**, written with no row at all (choose a pitch-class set as the key for that); **serialism beyond pitch**, whether durations, dynamics or registers follow a series too; or **combinatoriality**: in the Hexachord pads style it reads each track's own form, which is right, but it does not say that the two halves make all twelve notes together.

How the check reads the row

A form's places in sixteen bits

`Walk` goes through one form of the row, group by group, and remembers which of the twelve places have sounded. That is twelve yes-or-no answers, so it keeps them as the bits of a `u16`, one bit per place:

```

#[derive(Debug, Clone, Copy)]
struct Walk {
    /// One bit for each place in the form, from the first.
    used: u16,
    /// The place after the furthest one sounded.
    next: usize,
    errors: usize,
}

impl Walk {
    fn from(from: usize) -> Walk {
        Walk {
            used: (1 << from) - 1,
            next: from,
            errors: 0,
        }
    }

    fn heard(self) -> usize {
        self.used.count_ones() as usize
    }

    fn has(self, place: usize) -> bool {
        self.used & (1 << place) != 0
    }
    // ...
}

```

`Walk::from(3)` starts a walk whose first three places have sounded already, in the pattern before or as the shared notes of an elision: $(1 \ll 3) - 1$ is `0b111`. `Walk` is `Copy`, two machine words and a bit set, so trying a form costs nothing to set up. `count_ones` counts the places heard in one processor instruction on most machines. The `pitch-class sets` use the same idea, with twelve bits for the twelve notes.

When are new notes in order?

A group of n new notes is in order when they fill the next n places of the form, in any order among themselves. “The next places” can mean two things, and `window` tries both:

```

fn open(self, places: std::ops::Range<usize>, count: usize) -> u16 {
    places
        .filter(|&place| !self.has(place))
        .take(count)
        .fold(0, |bits, place| bits | 1 << place)
}

fn window(self, taken: u16, count: usize) -> u16 {
    let first_open = self.open(0..12, count);
    let ahead = self.open(self.next..12, count);
    match (taken & ahead).count_ones() > (taken & first_open).count_ones()
    {
        true => ahead,
        false => first_open,
    }
}

```

`open` builds a set of places with a `fold` that ORs in one bit per place. `first_open` is the first places still open, which catches up a note skipped just before. `ahead` is the places after the furthest note heard, which goes on after a skip. Whichever holds more of the group's notes wins.

Take P4 with D and B swapped: E F G C# F# Eb Ab **B D** C A Bb. At B, the form wants D, so B is out of place. At D, the first open place is D's own, so D is in order, and so is everything after it.

Now take P4 with D left out: E F G C# F# Eb Ab **B** C A Bb. B is out of place again. At C, the first open place is still D's, which never comes. But the place after the furthest note heard, B, is C's own, so C is in order, and so are A and Bb. Only B is marked. With `first_open` alone, every note after the missing D would be out of place.

One walk, two jobs

`take` takes in one group and finds what is out of place:

```

fn take(&mut self, form: &Form, group: &Group, mut tell: impl
FnMut(Misplaced)) {
    // ...
    for &played in &group.notes {
        let place = place_of(&played);
        let kind = if repeat(&played) {
            continue;
        } else if self.has(place) {
            SlipKind::EarlyRepeat
        } else if window & (1 << place) == 0 {
            SlipKind::RowOrder
        } else {
            continue;
        };
        self.errors += 1;
        tell(Misplaced {
            kind,
            played,
            place,
            wanted,
            heard: self.heard(),
        });
    }
    self.used |= fresh;
    if fresh != 0 {
        self.next = self.next.max(16 - fresh.leading_zeros() as usize);
    }
}

```

`let kind = if ... { continue; } else ...` works because `continue` never gives a value: its type is `!`, “never”, which fits in place of any type, so the other branches decide that `kind` is a `SlipKind`. A note that only repeats the note just played is skipped this way, and so is a note in its place. At the end, `16 - fresh.leading_zeros()` is the place after the highest bit in `fresh`, the furthest place this group reached.

The same `take` counts the problems while the readings are compared, and reports them for the reading that wins. It does both through the closure `tell`, which it calls for each note out of place. While comparing, the closure does nothing:

`walk.take(form, &groups[end - 1], |_| {})`. When reporting, it turns each note into a finding: `walk.take(form, group, |misplaced|`

`slips.push(misplaced.slip(form, key)))`. That second closure changes `slips`, a vector it borrows, so it is an `FnMut`, not just an `Fn`. Because `take` is generic over the closure (`impl FnMut`), the compiler builds a copy of `take` for each one, and the empty one costs nothing at all.

The best reading: dynamic programming

`read` has to choose where each form starts and ends, and which form it is.

`best[end]` is the fewest problems in the first `end` groups, and `came_from[end]` is the last stretch on the way there:

```
for start in 0..n {
  if best[start] == usize::MAX {
    continue;
  }
  // ...
  for from in 0..12 {
    for (index, form) in forms.iter().enumerate() {
      // ... may this form start at its place `from` here? ...
      let mut walk = Walk::from(from);
      for end in start + 1..=reach {
        walk.take(form, &groups[end - 1], |_| {});
        // ...
        let missing = if end == n { 0 } else { 12 - walk.heard() };

        let cost = best[start] + walk.errors + missing;
        if cost < best[end] {
          best[end] = cost;
          came_from[end] = Stretch {
            start,
            end,
            form: index,
            from,
          };
        }
      }
    }
  }
}
```

Each stretch is tried once for each form. The walk carries on from one `end` to the next instead of starting again, so a form tried on twenty groups costs twenty steps, not two hundred. A form may start at a later place (`from` above 0) only at the very start, carried over from the pattern before, or where the notes before it are the ones the group before shared (an elision). Forms from their first note are tried first, and only a strictly lower cost replaces a reading, which is how the ties go to them. Walking `came_from` back from the last group gives the forms in order.

`forms_of` lists the forms to try, skipping a form that is the same notes as one already listed, so Webern's palindrome is read with 24 forms, not 48. And `read_rows` runs `read` twice, once on all the notes and once per track, and keeps the cheaper:


```
if apart.iter().map(|(_, reading)| reading.cost).sum::() <
together.cost {
```

The same idea, keeping the best way to reach each point, shares out a chord's notes among the parts in the classical check ([Voice leading and part motion](#)).

Notes that start together: chunk_by

The groups come from the notes sorted by start, cut wherever the start changes:

```
for chunk in played.chunk_by(|a, b| a.note.start_tick() ==
b.note.start_tick()) {
```

`chunk_by` gives slices of neighbours for which the closure says "same group", without copying anything.

Testing it

`row.rs` checks the theory against the books: Schoenberg's four forms note by note against the published ones; every one of the 48 forms of the Lineup row named again by `label_of`; every line of Berg's matrix a P form and every column an I form, with the row's first note all down the diagonal; Webern's row a palindrome (R9 is P3); Berg's row, the All the intervals row and fifty random all-interval rows all-interval; the Lineup row combinatorial with I3 only; rows read from names, numbers and the compact form, and wrong ones refused with the right message; the same seed giving the same random row; and a row with a note written twice refused when read from a file:

```
#[test]
fn webern_is_a_palindrome() {
    let webern = RowPreset::named("Webern's Symphony").unwrap().row;
    let r9 = webern.form(RowLabel::new(RowForm::Retrograde, 9));
    assert_eq!(webern.label_of(r9).unwrap().to_string(), "P3");
    let mirror = RowPreset::named("Mirror row").unwrap().row;
    let backwards = mirror.form(RowLabel::new(RowForm::Retrograde, 0));
    assert_eq!(mirror.label_of(backwards).unwrap().to_string(), "P6");
}
```

`serial.rs` writes a melody in eighth notes, the four forms in turn filling the pattern, trichords as chords, the three series, and checks that no note runs past the end of the pattern. `library/rows.rs` checks that every style uses a row from the presets and writes its voices in any key; that Hexachord pads has all twelve notes in every bar; that

a song takes a row style, and that the style's chords replace the song's on their own channel instead of playing over them. One test joins the two halves of this chapter: it writes every style, on C and on F#, and runs the voice-leading check on it, which must read each one as forms of its row with no note out of order, so the styles and the check follow the same rules.

The tests of the rules in `voice_leading/serial.rs` write each rule and its exceptions as notes on Schoenberg's row: a row in order, chords from neighbouring notes, two notes swapped, a note left out, repeats and trills, an elision, a form left unfinished, a tune and chords sharing a form, a canon, octave doublings, tonal chords, and the classical rules switched off in rows and sets. Their names read as the rules:

```
two_notes_swapped_are_one_note_out_of_order ,  
a_note_may_be_repeated_or_trilled_but_not_brought_back_later ,  
one_form_may_end_on_the_note_the_next_begins_with .
```

Exercises on rows are in [Exercises: music theory in code](#).

Micromodes

Files: `crates/lineup-core/src/harmony/micromode.rs`, `crates/lineup-core/src/micromodal.rs`, `crates/lineup-core/src/library/micromodes.rs` and `crates/lineup-app/src/micromodes.rs`

[Pitch-class sets](#) give every group of notes a name. This chapter looks closely at the smallest groups that still have a character of their own, the groups of **three** notes, the way the Argentine composer **Francisco Kröpfl** taught his students to hear them. He called them **micromodes** (*micromodos*): twelve small “modes” of three notes, each with a colour you learn to recognise.

In the app, **Micromodes...** in the piano roll opens a window with Kröpfl’s twelve micromodes in his three families. Each is drawn on a clock, with its prime form in solfège (*do - do# - mi*), its intervals, its pitch-class set and its hour on Peter Schat’s Tone Clock; you can hear it, draw with it or use it as the key. Select some notes and the window lists the micromodes inside them. The Sets and Micromodes windows point at each other. The Library has a group **Micromodes (Kröpfl)** with seven styles, and **File > Demos > Micromodes** plays each of them as a short piece. The chord sheet can show any harmony as micromodes too (see [One harmony, four names](#)).

The music, briefly

Francisco Kröpfl (1931–2021) was born in Timișoara, Romania, and grew up in Argentina. He was a pioneer of electronic music in Latin America: in 1958 he founded the Estudio de Fonología Musical at the University of Buenos Aires, the first electronic music studio of a university on the continent. He was also a teacher of composition for decades, and his piano piece *Música '66* was written to try out the method he taught.

That method starts from a simple observation. Pick any three different notes. However you spell them, in whatever order and in whatever octaves you play them, the three notes have three intervals between them, and those intervals give the group a colour: C, D $\flat$ and E have the same tense, bittersweet sound whether you play them as a chord, as a melody, or spread over four octaves. Move the group up (D, E $\flat$, F $\sharp$) and it keeps its colour. Turn it upside down (C, B, A $\flat$, the same intervals going down) and the colour is still the same family of sound.

So Kröpfl counted three notes as **the same micromode** when they are:

- the same notes in another order or octave, or spelled another way (C# and D $\flat$ are one note here),
- moved up or down (transposed), or
- turned upside down (inverted).

These are exactly the rules of pitch-class sets. Of the 220 ways to choose three notes out of twelve, the rules leave only **twelve** different micromodes. Repeat one of them through a passage and the passage takes its colour; Pablo Cetta, a composer and theorist who studied Kröpfl's music, describes it as "un 'color' o timbre global particular y reconocible", a particular, recognisable overall colour or timbre.

The table

Each micromode is written in its **prime form**: its most compact shape, starting on C (*do*), the same prime form as in the Sets window. This is Pablo Cetta's table, with the numbers Allen Forte gave the same twelve groups in 1973:

	Micromode	Prime form	Forte	Intervals	Forms
1	Menor 1	do - do# - re	3-1 [012]	two minor seconds, a major second	12
2	Menor 2	do - do# - re#	3-2 [013]	minor second, major second, minor third	24
3	Menor 3	do - do# - mi	3-3 [014]	minor second, minor third, major third	24
4	Menor 4	do - do# - fa	3-4 [015]	minor second, major third, fourth	24
5	Menor 5	do - do# - fa#	3-5 [016]	minor second, fourth, tritone	24
6	Mayor 1	do - re - mi	3-6 [024]	two major seconds, a major third	12
7	Mayor 2	do - re - fa	3-7 [025]	major second, minor third, fourth	24

	Micromode	Prime form	Forte	Intervals	Forms
8	Mayor 3	do - re - fa#	3-8 [026]	major second, major third, tritone	24
9	Mayor 4	do - re - sol	3-9 [027]	major second, two fourths	12
10	Acorde disminuido	do - mib - solb	3-10 [036]	two minor thirds, a tritone	12
11	Acorde menor-mayor	do - mib - sol	3-11 [037]	minor third, major third, fourth	24
12	Acorde aumentado	do - mi - sol#	3-12 [048]	three major thirds	4

“Fourth” here means a perfect fourth *or* a perfect fifth: in a set, an interval and its inversion count as one (C up to G is a fifth, G up to C a fourth; the pair C and G is the same either way). “Forms” counts how many different groups of three notes are that micromode: the prime form and its inversion on each of the twelve notes. Most have 24; those that are their own inversion have 12; the augmented triad only 4, because moving it up a major third gives the same three notes again. The forms add up to 220, every group of three there is.

Three families, and their names

Kröpfl sorted the twelve into three families. Pablo Jaureguiberry, who analysed the music of Kröpfl’s student Diana Rud, gives the rules:

- “Los micromodos **menores** son todos aquellos que están compuestos por al menos un intervalo de segunda menor”: the **minor** micromodes are all those with at least one minor second (a semitone). There are five, Menor 1 to 5.
- “Los **mayores** tienen todos al menos una segunda mayor y a su vez carecen de semitonos”: the **major** ones all have at least one major second and no semitone. There are four, Mayor 1 to 4.
- “Los restantes contienen al menos dos terceras cada uno”: the rest have at least two thirds each. They are the three triads, and they are named after them: the **diminished**, the **minor and major** (*menor-mayor*), and the **augmented** chord.

The names come from the prime form, as Cetta explains: the first nine start with a second, minor (*do - do#*) or major (*do - re*), and the number counts up as the third

note rises. So Menor 3 is “the third group that starts with a minor second”, *do - do# - mi*.

One micromode deserves a second look: **Acorde menor-mayor**. Its prime form, C E $\flat$ G, is a minor triad. Turned upside down from C it is C, A and F: an F major triad. Minor and major triads are mirror images of each other, so they are one micromode, and its name says both. Every chord of a I-IV-V-I progression is the same micromode; what tonal music does with them is a matter of *which* forms, not which colour.

Finding the micromode of three notes by hand

Cetta gives two examples of Menor 3: E, F and A $\flat$, and F, D and G $\flat$.

1. **E F A $\flat$** . Put the notes in order inside one octave, as close together as they go: E F A $\flat$ spans four semitones, the smallest span. Move the group so it starts on C: C C $\sharp$ E. That is *do - do# - mi*, the prime form of Menor 3.
2. **F D G $\flat$** . In order and as close as possible: D F G $\flat$, a span of four semitones. From C: C E $\flat$ E, which is *do - mib - mi*, not in the table. Turn it upside down: the intervals 3 then 1 become 1 then 3, C C $\sharp$ E. Menor 3 again.

That is the whole algorithm: the **normal form** (the closest order), moved to C, compared with its inversion, the more compact of the two winning. The Sets window already does it, so Ferrous Lineup finds the micromode of three notes by asking for their prime form and looking it up in the table.

The same twelve, three times

Kröpfl was not the only one to find these twelve. In *The Structure of Atonal Music* (1973), the American theorist **Allen Forte** listed every set class, and his twelve trichords, 3-1 to 3-12, are the micromodes in the same order. The Dutch composer **Peter Schat** found them again and arranged them as the twelve hours of his **Tone Clock** (*The Tone Clock*, 1993): hour I is Menor 1, hour III is Menor 3, hour XII the augmented triad.

The three did not mean the same thing by them, though. Marcela Pavia, comparing Forte and Kröpfl, describes Forte's classification as mathematical, and Kröpfl's as built on how we *perceive* groups of notes, after the Gestalt psychologists; she finds that the two give very different results when you use them to analyse music. Forte counts every subset of every chord; Kröpfl listens for the groups the music makes you hear as a unit. Ferrous Lineup shows both: the Forte number beside every micromode, and the micromodes beside every Forte set.

Bigger sets are made of micromodes

A set of n notes holds $n(n-1)(n-2)/6$ groups of three: 4 for a four-note chord, 10 for a pentatonic scale, 35 for a seven-note scale. Counting which micromodes they are gives the set's **micromode content**, a fingerprint of its colours. Some examples, all checked by the tests or worked out with the same functions:

- **The dominant seventh**, C E G B $\flat$: C E G is a major triad (Acorde menor-mayor), E G B $\flat$ a diminished triad, C E B $\flat$ is Mayor 3 (the chord without its fifth) and C G B $\flat$ is Mayor 2. One of each.
- **The pentatonic scale**, C D E G A (5-35): Mayor 2 four times, Mayor 4 three times, Acorde menor-mayor twice (C E G and A C E) and Mayor 1 once (C D E). No semitone, so no minor micromode at all, which is why the pentatonic scale sounds so smooth.
- **The diatonic scale**, the seven notes of C major (7-35): Mayor 2 $\times 8$, Acorde menor-mayor $\times 6$ (the three major and three minor triads), Mayor 4 $\times 5$, Menor 2 $\times 4$, Menor 4 $\times 4$, Mayor 1 $\times 3$, Menor 5 $\times 2$, Mayor 3 $\times 2$ and one diminished triad (B D F). No Menor 1, Menor 3 or augmented triad: you cannot play those with the white keys alone.
- **The whole-tone scale** (6-35): Mayor 3 $\times 12$, Mayor 1 $\times 6$ and the augmented triad twice. Nothing else, because every interval in it is even.
- **C C $\sharp$ D $\sharp$ E** (4-3), from Jaureguiberry's analysis of Diana Rud's *Tres amores* (1986): two Menor 2 and two Menor 3. Rud builds her four-note chords so that each bar keeps three notes of the bar before and adds a new one, and the micromodes inside them change bit by bit.
- **C C $\sharp$ F $\sharp$ G** (4-9), from Cetta's analysis of *Música '66*: four groups of three, all Menor 5. Kröpfl sets these next to Menor 3 in the piece.

This is how the two windows link. A micromode *is* a pitch-class set, one of the twelve with three notes, so the Sets window marks those twelve with their micromode's name; and a larger set is *made of* micromodes, so the Sets window lists them under its details ("Micromodes inside, in its 35 groups of three"). In the Micromodes window, the notes you select are counted the same way.

A row made of one micromode: Webern's Concerto

Composers before Kröpfl had heard the same thing. Arnold Schoenberg's *Piano Piece*, Op. 11 No. 1 (1909), opens with B, G $\sharp$ and G: Menor 3. Anton Webern went further. The row of his *Concerto for nine instruments*, Op. 24 (1934), is made of four groups of three (see [Twelve-tone rows and their rules](#) for rows):

B B $\flat$ D | E $\flat$ G F $\sharp$ | A $\flat$ E F | C C $\sharp$ A

Every group is Menor 3, and each is a form of the first:

group	notes	intervals	form of the first group
1	B B $\flat$ D	down 1, up 4	itself
2	E $\flat$ G F $\sharp$	up 4, down 1	backwards and upside down (RI)
3	A $\flat$ E F	down 4, up 1	backwards (R)
4	C C $\sharp$ A	up 1, down 4	upside down (I)

At the start of the Concerto the four groups are played by oboe, flute, trumpet and clarinet, one after another, so you hear the same colour four times in four instruments. Schat built harmonies the same way, from forms of one hour that together use all twelve notes; in his terms, Webern's row is made of hour III.

Webern did not think in micromodes: the word came decades later. The Concerto is a twelve-tone piece, and Webern built its row from four forms of one group; "Menor 3" and "3-3" are later names for that group, the way a theorist would describe it. Cetta notes that Kröpfl, like the composers of Schoenberg's circle, favoured groups of three notes, which is why the two meet here.

The row is one of the famous rows of the Row window, **Webern's Concerto**, and the style **Webern's trichords** plays it: the row as a melody leaping over three octaves, with each group of three as a chord under it. It is in the Library's **Twelve-tone rows** group and at the end of **File > Demos > Micromodes**. The rhythm is Ferrous Lineup's, not Webern's.

Only music in the public domain is copied into Ferrous Lineup. Webern died in 1945, so his music is in the public domain wherever copyright lasts 70 years after the author's death, and only the twelve notes of the row are used, never his score. Kröpfl died in 2021 and his students are alive, so their music is still under copyright, and none of it is in the app: the seven micromode styles were written for Ferrous Lineup.

Music from one micromode

How do you write music with one micromode? Alejandro Martínez describes the micromode as "elemento mínimo de construcción melódico/armónica", the smallest building block of melody and harmony alike. The chords are forms of the micromode, and so are the groups of the melody, so the whole passage has one colour. The

question is which form comes next. Ferrous Lineup offers two answers, in `micromodal.rs`.

Moving every note the same distance

The simplest is a **sequence**: move all three notes up by the same interval each time. *Whole-tone haze* moves Mayor 1 up a whole tone every bar: C D E, D E F#, E F# G#... It never leaves one whole-tone scale, the sound of Debussy's *Voiles*. *Augmented drift* slides the augmented triad down a semitone at a time; after four chords the fifth has the notes of the first, because there are only four augmented triads.

Keeping two notes, moving one

The smoother way is to keep **two** notes of each group and move only the third, so that each group grows out of the last. For the triads, this gives a famous set of moves. From C major (C E G):

move	keeps	moves	gives
P (parallel)	C and G	E down to E $\flat$	C minor
L (leading-tone exchange)	E and G	C down to B	E minor
R (relative)	C and E	G up to A	A minor

These are the three moves of **neo-Riemannian theory**, which theorists use to explain the chromatic chord changes of Wagner and Liszt. The style *Triads, two notes kept* walks this way: C minor, C major, E minor, E major, C# minor, D $\flat$ major, B $\flat$ minor, B $\flat$ major...

The same walk works for every micromode, with a surprise. Three of them cannot get out of a small corner of the twelve notes:

- **Mayor 1** (C D E) and **Mayor 3** (C D F#) have only even intervals: 2, 4 and 6 semitones. A form that keeps two of their notes must add a third at an even distance from them, so it stays in the same **whole-tone scale**. The walk reaches 6 notes, never 12.
- The **diminished triad** has only intervals of 3 and 6 semitones, so it stays in one **diminished seventh chord**: 4 notes.
- The **augmented triad** has only intervals of 4 semitones. Two of its notes fix the third, so no other form shares two notes with it. Its walk moves it up a semitone instead.

Every other micromode has an odd interval somewhere, and its walk goes round all twelve notes. The tests check each of these numbers.

Which form to keep

Several forms share two notes with the current one (for Menor 3 on C C# E there are three: A C C#, C D# E and C# E F), so the walk needs a rule to pick one. In order:

1. the form whose **new note** has not been heard for the longest time, so the music goes round all the notes it can reach,
2. then the form itself used least recently, so a walk shut inside a few notes (Mayor 1, the diminished triad) still goes round all its forms instead of rocking between two,
3. then the smallest movement of the notes that change,
4. and finally the set's bits, so the choice is always the same.

Textures

A walk is a list of forms. A **texture** says how one instrument plays them: as **chords** (the shared notes stay where they are and the changing note moves to the nearest place: smooth voice leading), as chords struck again and again (**pulse**), as a **line** (the new note first, then the others), as an **arpeggio** (up and down), or as a **bass** (the root of a triad, otherwise the first note of the normal form). Every instrument of a style plays the same walk, each in its own texture, so the bass, the chords and the tune always share their three notes.

A line plays the note that just arrived first, so you hear what changed, then the two that stayed. `write_forms` splits the form's notes in two with `partition`, which sends each item to one of two collections by a yes-or-no closure, in one pass:

```
let (new, kept): (Vec<PitchClass>, Vec<PitchClass>) =  
    normal.into_iter().partition(|p| !old.contains(*p));  
new.into_iter().chain(kept).collect()
```

`old` is the form before; the notes not in it are new. `chain` then joins the two lists again, new notes first.

The seven styles:

style	micromode	walk	sound
Menor 3 colour	Menor 3	keep two notes	chords, tune and bass in one colour
Viennese points	Menor 5	keep two notes	single notes leaping over three octaves over soft chords
Cluster pulse	Menor 1	keep two notes	clusters creeping up a semitone at a time, over a techno beat
Whole-tone haze	Mayor 1	up a whole tone	arpeggios in 3/4
Fourths groove	Mayor 4	keep two notes	sus chords over a neo-soul beat
Triads, two notes kept	menor-mayor	keep two notes	P, L and R moves
Augmented drift	aumentado	down a semitone	four augmented triads, round and round

A demo plays one pass from the micromode on C, then upside down, then from a tritone away (F#), then upside down from there, and the first pass again. Click a style in the Library while the song plays and the whole song goes over to it: its chords and bass are replaced by the walk, the patterns start from those four places in turn, and the key becomes the micromode itself, a pitch-class set, so the harmony keys play its forms.

The micromodes in Rust: a table in a static

A micromode is data: a name, a family, a prime form, a line of text. The twelve are a `static` array of a plain struct:

```

/// One of the twelve micromodes.
#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash)]
pub struct Micromode {
    /// From 1 to 12: its place in the table, which is also its Forte
    /// number (3-`number`) and its hour on Peter Schat's Tone Clock.
    pub number: u8,
    pub family: MicromodeFamily,
    /// Kröpfl's name, in Spanish, as his students use it: "Menor 3".
    pub name: &'static str,
    /// The name in English.
    pub english: &'static str,
    /// The prime form, from C.
    pub prime: PitchSet,
    /// The prime form in solfège, as in Cetta's table: "do - do# - mi".
    pub solfege: &'static str,
    /// Where you hear it: a chord or scale it belongs to, or music that
    /// is built from it.
    pub about: &'static str,
}

```

```

static MICROMODES: [Micromode; 12] = [
    Micromode {
        number: 1,
        family: MicromodeFamily::Minor,
        name: "Menor 1",
        english: "Minor 1",
        prime: prime(0b0000_0000_0111),
        solfege: "do - do# - re",
        about: "Three notes a semitone apart: the smallest cluster, a
chromatic scale in \
                miniature.",
    },
    // ...
];

```

`prime(0b0000_0000_0111)` is the set {C, C#, D} written as the bits of a `u16`, one bit per note, with C as the lowest bit (see [A set of pitch classes in twelve bits](#)). `prime` is a `const fn`, so the whole table is built at compile time and costs nothing when the program starts. Every field is `Copy` (numbers, enums, `&'static str` and `PitchSet`, which is a `u16`), so `Micromode` derives `Copy` too, but the rest of the code passes `&'static Micromode` around: a reference into the table, which can never dangle because the table lives as long as the program. Kröpfl's names are in Spanish in both languages of the app; in English the window adds the English name in brackets, "Menor 3 (minor 3)".

Finding the micromode of three notes reuses the set theory:

```
pub fn of(set: PitchSet) -> Option<&'static Micromode> {
    if set.len() != 3 {
        return None;
    }
    let prime = set.prime_form();
    MICROMODES.iter().find(|m| m.prime == prime)
}
```

Checking a table with tests

A hand-written table can have typos. The tests check it from three directions, none of which trusts the table:

- **Against Forte.** The set classes, filtered to three notes, must be the twelve prime forms in the same order, named 3-1 to 3-12.
- **Against counting.** A loop over all 4,096 sets (every `u16` from 0 to 4,095) checks that exactly the sets of three notes have a micromode, that there are 220 of them, and that each micromode's forms match how often it was found.
- **Against the rules.** The families are worked out again from the intervals, with Jaureguiberry's rules, by a function that only the tests can see:

```
#[cfg(test)]
fn of_intervals(vector: [u8; 6]) -> MicromodeFamily {
    if vector[0] > 0 {
        MicromodeFamily::Minor
    } else if vector[1] > 0 {
        MicromodeFamily::Major
    } else {
        MicromodeFamily::Chord
    }
}
```

`#[cfg(test)]` on a single function compiles it only for `cargo test`. Without it, `cargo clippy` warns that the function is never used, because only the tests call it. The solfège is checked too: each syllable is turned back into a number ("do#" is 1, "mib" is 3) and the notes must be the prime form.

Intervals in words

`vector[0]` counts minor seconds and `vector[1]` major seconds: it is the interval vector of [Pitch-class sets](#). The same vector gives the intervals in words, for the window:

```

pub fn intervals(&self) -> String {
    const NAMES: [(&str, &str); 6] = [
        ("minor second", "minor seconds"),
        ("major second", "major seconds"),
        ("minor third", "minor thirds"),
        ("major third", "major thirds"),
        ("perfect fourth or fifth", "perfect fourths or fifths"),
        ("tritone", "tritones"),
    ];
    let vector = self.prime.interval_vector();
    let mut parts = Vec::new();
    for (class, &count) in vector.iter().enumerate() {
        let (one, many) = NAMES[class];
        // Whole phrases, as a Spanish number goes with its noun.
        match count {
            0 => {},
            1 => parts.push(crate::i18n::tr_string(format!("a {one}"))),
            2 => parts.push(crate::i18n::tr_string(format!("two
{many}"))),
            _ => parts.push(crate::i18n::tr_string(format!("three
{many}"))),
        }
    }
    crate::i18n::list(&parts)
}

```

The plurals are a pair of strings instead of `format!("{name}s")`, because English plurals are not always an added “s”: “two perfect fourths or fifths”, not “two perfect fourth or fifths”. And each phrase is translated whole (“a minor second”, “two minor seconds”), because in Spanish the number changes with the noun: *una segunda menor*, *dos segundas menores* (see [Translating: the English text is the key](#)). `list` then joins the phrases:

```

pub fn list(items: &[impl AsRef<str>]) -> String {
    match items.split_last() {
        None => String::new(),
        Some((last, [])) => last.as_ref().to_string(),
        Some((last, rest)) => {
            let rest: Vec<&str> =
rest.iter().map(AsRef::as_ref).collect();
            trf("{} and {}", &[&rest.join(", "), &last.as_ref()])
        }
    }
}

```

`split_last` gives the last item and the slice before it, or `None` for an empty list. `Some((last, []))` is a **slice pattern**: `[]` matches only an empty slice, so this arm is a list of one. Then `["a minor second", "a minor third", "a major third"]` becomes “a minor second, a minor third and a major third”, with the “and” (or the

Spanish “y”) in the right place and no index arithmetic that could go wrong. `items: &[impl AsRef<str>]` takes a slice of `String`s or of `&str`s alike.

Counting groups of three

`trichords` lists every group of three notes of a set, with three nested loops whose indexes always go up, so no group is listed twice:

```
pub fn trichords(set: PitchSet) -> Vec<PitchSet> {
    let notes: Vec<PitchClass> = set.iter().collect();
    let mut trios = Vec::new();
    for a in 0..notes.len() {
        for b in a + 1..notes.len() {
            for c in b + 1..notes.len() {
                trios.push([notes[a], notes[b],
notes[c]].into_iter().collect());
            }
        }
    }
    trios
}
```

`[notes[a], notes[b], notes[c]].into_iter().collect()` turns an array of three pitch classes into a `PitchSet`: `PitchSet` implements `FromIterator<PitchClass>`, so `collect` knows how to build one, and the return type `Vec<PitchSet>` tells it which collection to build.

`micromode_content` counts them into a `[u16; 12]`, one counter per micromode. To show them, `micromodes_in` keeps the ones found and sorts them, the most common first:

```
pub fn micromodes_in(set: PitchSet) -> Vec<(&'static Micromode, u16)> {
    let counts = micromode_content(set);
    let mut found: Vec<(&'static Micromode, u16)> = MICROMODES
        .iter()
        .zip(counts)
        .filter(|(_, count)| *count > 0)
        .collect();
    found.sort_by_key(|(m, count)| (std::cmp::Reverse(*count), m.number));
    found
}
```

`sort_by_key` sorts from the smallest key up, and here the *largest* count comes first. `std::cmp::Reverse` wraps a value and compares it the other way round, so `Reverse(8)` comes before `Reverse(6)`. The key is a **tuple**, and tuples compare item by item: first the counts (reversed), and only when two counts are equal, the

micromode's number. So equal counts keep the table's order, and the list is the same every time it is drawn.

The walk: choosing with a tuple

The walk's rule, "the freshest new note, then the form used least recently, then the smallest move", is one `min_by_key` with a tuple of four:

```
members
  .iter()
  .copied()
  .filter(|&set| set != current && shared(set) == most)
  .min_by_key(|&set| {
    let new = PitchSet::from_bits(set.bits() & !current.bits());
    // The new note heard most recently decides; never heard is best.
    let freshest = new
      .iter()
      .map(|p| heard[usize::from(p.semitone())].map_or(0, |i| i +
1))
      .max()
      .unwrap_or(0);
    // How far the moving notes go, each to its nearest new note.
    // ...
    // Then the form itself heard least recently, so a walk shut in
    // a few notes still goes round all its forms.
    let used = before.iter().rposition(|&f| f == set).map_or(0, |i| i
+ 1);
    (freshest, used, moves, set.bits())
  })
  .unwrap_or(current)
```

A few things to notice:

- `members` is every form of the micromode (24, 12 or 4 of them). `shared(set)` counts the notes a form has in common with the current one, with the bits: `set.bits() & current.bits()` keeps the notes in both. `set.bits() & !current.bits()` keeps the notes of `set` that are *not* in `current`: the new note.
- `heard` is an array of twelve `Option<usize>`, one per pitch class: `None` if the note has not been played yet, or `Some(i)` if it was last played in form number `i`. `map_or(0, |i| i + 1)` turns "never" into 0, the smallest key, so a note never heard always wins, and a note heard long ago beats one heard just now.
- `rposition` searches a slice **from the end** and gives the index of the last item that matches: when the form was last used. Again "never" becomes 0.
- Because the tuple compares item by item, the order of the four items *is* the order of the rules. Changing the rules means reordering a tuple, not rewriting a

chain of `if` s.

How many forms does one pass need? A style says how many steps each form lasts, and the pattern has a length in steps; the count rounds up, so a last form that only half fits still gets written:

```
let count = length.div_ceil(self.steps.max(1)).max(1);
```

`div_ceil` is division that rounds up: `16.div_ceil(6)` is 3, where `16 / 6` is 2.

Styles take over the harmony

A micromode style, like a twelve-tone style, has no chords to follow. When you click it while the song plays, it *replaces* the song's harmony: it silences the chords and the bass it does not play, and writes its own voices on the right channels. The row styles do the same, so both kinds of style call one function, from [Twelve-tone rows and their rules](#):

```
pub(super) fn take_over_harmony(
    project: &mut Project,
    uses: &dyn Fn(&str) -> bool,
    rhythm: Option<&Rhythm>,
    mut write: impl FnMut(&mut Project, usize, usize, Option<usize>),
) -> RhythmChange {
```

The two closures are passed in two different ways, on purpose:

- `uses` only answers a question ("does the style play this channel?"), so it is an `&dyn Fn`: a **trait object**, a reference to some closure whose type is decided at run time. One compiled copy of `take_over_harmony` works for every closure.
- `write` is called once per pattern with `&mut Project`, and does the writing. It is an `impl FnMut`: the compiler makes a copy of `take_over_harmony` for each closure type passed in (one for the row styles, one for the micromode styles), which is a little faster and allows the closure to change what it captured. `FnMut` is the loosest bound that can still be called many times.

The micromode styles call it like this:

```

pub fn use_micromode_style(project: &mut Project, style: &MicromodeStyle)
-> RhythmChange {
    let key = project.scale.root;
    let uses = |name: &str| style.voices.iter().any(|voice| voice.channel
== name);
    let change = take_over_harmony(
        project,
        &uses,
        style.rhythm(),
        |project, pattern, n, keys| {
            let start = DEMO_SECTIONS[n % DEMO_SECTIONS.len()].0;
            write_voices(project, pattern, style, key, start, keys);
        },
    );
    if change.harmony > 0 {
        project.scale = style.scale(key);
    }
    change
}

```

`n` counts the patterns written so far, so the first pattern of the song starts from the micromode, the next upside down, the next a tritone away, and so on: the song does not repeat itself. The closure is given `project` as an argument instead of capturing it: `take_over_harmony` already holds `&mut Project`, and Rust allows only one mutable borrow at a time, so the project is lent to the closure for each call.

The demos are arranged like the twelve-tone demos: `section_patterns` makes one pattern per section and fills it with a closure, and `arrange_sections` lays them out on the playlist, with the drums coming in after the first section. `micromode_demo` and `row_demo` both use them, each passing its own closure.

The window: a clock drawn with egui's painter

The twelve notes sit round a circle like the hours of a clock, C at the top, and the micromode's three notes are joined into a triangle. The shape is the micromode: moving the group turns the triangle round the clock, and turning it upside down mirrors it. That is why Schat called the trichords hours.

```

/// The point for pitch class `n` on a clock face: C at the top, going
/// round clockwise a semitone at a time.
fn on_clock(center: Pos2, radius: f32, n: u8) -> Pos2 {
    let angle = f32::from(n) / 12.0 * TAU;
    center + Vec2::new(angle.sin(), -angle.cos()) * radius
}

```

`TAU` is 2π , a whole turn, so pitch class `n` is `n` twelfths of a turn. In maths, angle 0 points right and angles go anticlockwise; here 0 is at the top, going clockwise. Using `sin` for x and `-cos` for y does both: at 0, x is 0 and y is -1, which is *up* on a screen, because screen y grows downwards.

`clock` then asks for a square of the window with `ui.allocate_exact_size`, takes a `painter_at` that rect (anything drawn outside it is clipped), and draws: the circle with `circle_stroke`, the triangle with `Shape::convex_polygon` (filled with the accent colour at a quarter strength, `gamma_multiply(0.25)`), a dot for every note, bigger for the three in the group, and the names round the outside. The buttons of the list, each a tiny clock and a name, are drawn the same way: `allocate_exact_size` with `Sense::click()` gives a `Response` that knows about hovering and clicks, like a real button.

Two windows that point at each other

Neither window calls the other. Each returns a list of **actions**, as the other windows of the piano roll do:

```
/// Something done in the micromode window.
pub enum MicroAction {
    /// Make this group the key.
    UseAsKey(Scale),
    /// Play these notes now.
    Hear(Vec<Note>),
    /// Show this set in the set browser.
    ShowSet(PitchSet),
}
```

The Sets window has two actions of its own for this, `ShowMicromode` and `OpenMicromodes`. The piano roll owns both windows, so it handles the actions: `ShowSet` calls the Sets window's `pick` and opens it; `ShowMicromode` calls the Micromodes window's `pick`. Each window only knows how to show a set it is given, and the piano roll connects them. (**Draw with it** needs no action: both windows are handed the piano roll's `ChordStamp` as a `&mut`, and set it directly.)

`pick` shows not only the micromode but the very form that was clicked. Click Menor 3 under Webern's row in the Sets window, and the Micromodes window opens on B $\flat$ D, not on C C# E:

```
pub fn pick(&mut self, form: PitchSet) {
    if let Some(micromode) = Micromode::of(form) {
        self.micromode = micromode;
        let (n, inverted) = micromode.prime.moved_to(form).unwrap_or((0,
false));
        self.transposition = n;
        self.inverted = inverted;
        self.raise = true;
    }
}
```

`moved_to`, from [Pitch-class sets](#), finds how to move the prime form onto the form: up `n` semitones, upside down or not.

A window shown from another must come to the front, or it may stay *behind* the window you clicked in. A window that opens is kept in front by `Front` (see [A window that opens in front](#)); one that was open already, but behind, is raised by the flag `raise`: `pick` sets it, and the next frame reads it and clears it in one go with `std::mem::take`, which returns the value and leaves `false` behind:

```
if std::mem::take(&mut self.raise) {
    ctx.move_to_top(egui::LayerId::new(egui::Order::Middle, window_id()));
}
```

The Sets window does the same with a `reveal` flag, and also scrolls its long list to the class it was asked to show: `response.scroll_to_me(Some(egui::Align::Center))` on the line of that class, for one frame only.

Spelling the triads as triads

C, E $\flat$ and F $\sharp$ and C, E $\flat$ and G $\flat$ are the same three keys, but only one is a diminished triad on paper: C E $\flat$ G $\flat$, a stack of two minor thirds. The three chord micromodes are spelled as the chords they are. The `NoteNames` trait, which writes notes either in the key's spelling or in plain sharps or flats, has a second method for this:

```
pub trait NoteNames: Copy {
    fn note_name(self, pitch_class: PitchClass) -> NoteName;
    /// The notes of a chord, from its root up.
    fn chord_names(self, chord: Chord) -> Vec<NoteName>;
}
```

`Micromode::chord` finds the triad for a form (the root is the note of the normal form from which the other two make a known chord), and the window writes its notes with

`chord_names` . The augmented triad has no real root (each of its notes is a major third from the next), so it is named from the first note of its normal form.

Sources

Kröpfl never published his theory; as Jaureguiberry writes, “esta propuesta nunca fue publicada por su autor”. It is known from his students and from analyses of their music. These are the sources for this chapter and for the texts in the app:

- Pablo Cetta, “*La tercera es la vencida* de Francisco Kröpfl y Oscar Steimberg”, *Cuadernos de Análisis y Debate sobre Músicas Latinoamericanas Contemporáneas* 1 (2018), pp. 8–36. The table of the twelve micromodes (Ejemplo 1) with Forte’s numbers, the prime forms in solfège, the examples E F A $\flat$ and F D G $\flat$, the analysis of *Música ’66* (Menor 3 and Menor 5, the set 4-9) and of the witch’s song in Kröpfl’s chamber opera (2011), built mainly on Menor 3. Online at the repository of the Universidad Católica Argentina.
- Pablo Jaureguiberry, “*Tres amores* de Diana Rud: otras particularidades de los micromodos de Francisco Kröpfl” (Universidad Nacional de Rosario, 2016), pp. 499–510. The definition, the three families and their rules, Menor 3 as a favourite of Kröpfl’s students, and four-note sets made of two Menor 2 and two Menor 3.
- Alejandro Martínez, “Algunos aspectos de los micromodos de Francisco Kröpfl y su contextualización histórica”, in *Actas de la XXI Conferencia de la Asociación Argentina de Musicología* (2015), pp. 399–415. The micromode as the smallest unit of melody and harmony, its use by Argentine composers of several generations, and its place beside Forte’s sets, Gilbert’s trichordal analysis and Schat’s Tone Clock.
- Marcela Pavia, “La estructura de la música atonal: dos enfoques”, *CodeXXI* 1 (1998), pp. 199–214. Kröpfl’s classification, built on perception, compared with Forte’s, built on mathematics.
- Allen Forte, *The Structure of Atonal Music* (Yale University Press, 1973). The trichords 3-1 to 3-12.
- Peter Schat, *The Tone Clock* (Harwood Academic Publishers, 1993). The twelve hours.

The musical examples by other composers are only named, never copied, except for the twelve notes of Webern’s row.

Testing it

`micromode.rs` checks the table and the sources: the twelve match Forte's trichords in order; all 4,096 sets give 220 groups of three, each in exactly one micromode, with 12, 24 or 4 forms; the families follow from the intervals by Jaureguiberry's rules; the solfège spells each prime form; names are found in Spanish and English, in any case; Cetta's E F A $\flat$ and F D G $\flat$, Schoenberg's B G $\sharp$ G and the four groups of Webern's row are Menor 3; C C $\sharp$ D $\sharp$ E holds two Menor 2 and two Menor 3; 4-9 holds four Menor 5; the dominant seventh holds one each of Mayor 2, Mayor 3, the diminished triad and menor-mayor; and the whole-tone scale holds only Mayor 1, Mayor 3 and the augmented triad.

`micromodal.rs` checks the walk and the writing: every step keeps two notes; the walk reaches all twelve notes, or 6 for Mayor 1 and Mayor 3, or 4 for the diminished triad, with no form repeated in its first four steps; the triads' moves are P, L or R; a sequence of Mayor 1 stays in one whole-tone scale; chords keep their shared notes on the very same keys; every texture writes notes of the right forms and never past the pattern's end; the bass plays roots.

```
#[test]
fn the_bass_plays_the_root_of_a_triad() {
    assert_eq!(bass_note(set(&[9, 0, 4])), PitchClass::A);
    assert_eq!(bass_note(set(&[11, 2, 5])), PitchClass::B);
    assert_eq!(bass_note(set(&[0, 1, 4])), PitchClass::C);
    assert_eq!(bass_note(set(&[11, 0, 3])), PitchClass::B);
}
```

A C E is A minor, so its bass is A; B D F is B diminished; C C $\sharp$ E is no triad, so its bass is the first note of its normal form, C; and B C E $\flat$ is no triad either, starting on B.

`library/micromodes.rs` checks that every style plays only its micromode, in any key and from every section's start, with the key becoming the micromode; that the first form starts on the key note; and that a song takes a micromode style. The app's tests check that chord micromodes are spelled as chords (C E $\flat$ G $\flat$), that ► **Hear it** plays the three notes from middle C, and that picking a form shows how it was moved.

Exercises on micromodes are in [Exercises: music theory in code](#).

Voice leading and part motion

Files: `crates/lineup-core/src/harmony/voice_leading.rs`, `crates/lineup-core/src/harmony/voice_motion.rs` and `crates/lineup-app/src/piano_roll.rs`

The chord lane names each chord you write. That is half of harmony. The other half is **voice leading**: how each part moves from its note in one chord to its note in the next. A part is one line of notes, such as a singer, an instrument, or one line in a piano's chords. Two pieces can use the same chords and sound very different, smooth or clumsy, because their parts move differently.

Tick **Voice leading** next to **Ghost notes** in the piano roll, and it does two things a harmony teacher does. It marks where the parts break the rules of classical part writing, the red pen; and it shows how every part moves at every chord change, good or not, which is most of what there is to learn. This chapter covers the theory, the rules and their exceptions, and the code: dynamic programming to split a track into parts, `Option` chains to find parallel fifths, and a cache so the check runs only when something changed.

The music, briefly

Classical harmony courses teach voice leading with four-part writing: soprano, alto, tenor and bass, as in Bach's chorales. They give a few rules that keep every part a line of its own. The check follows the best known three: **no parallel fifths or octaves, the leading tone rises, a seventh falls**. They are explained one by one in [The rules](#) below. Two more ideas describe every move, good or bad.

Steps, skips and leaps

How far a part moves has a name:

Move	Interval	Semitones	Example
Held	the same note	0	G3 to G3
Step	a second	1 or 2	F4 to E4
Skip	a third	3 or 4	B3 to G3
Leap	a fourth or more	5 or more	G2 to C3

The inner parts of a chorale move mostly by step or hold their notes. The bass leaps more, because it moves from root to root: G2 up a fourth to C3 at a cadence is one of the commonest bass moves there is.

Motion between two parts

Two parts go together in one of four ways:

Motion	The two parts	Example
Parallel	move the same way and keep the same interval	thirds going to thirds
Similar	move the same way, to another interval	a sixth going to a fifth, both down
Contrary	move opposite ways	the bass up, the top down
Oblique	one moves, the other holds	a held note under a moving one

Part writing likes **contrary** and **oblique** motion between the outer parts, the top line and the bass, because the ear follows those two most. Moving opposite ways keeps them two lines. Parallel thirds and sixths are fine anywhere, and they sound sweet. Parallel fifths and octaves are what the check marks, because they make two parts sound like one.

“The same interval” means the same interval **number** as the key spells it, not the same number of semitones. From E4 over G3 to F4 over A3, the upper part goes up a minor second and the lower a major second. E over G is a major sixth and F over A a minor sixth: a sixth both times, so that is parallel motion, parallel sixths. A harmony teacher counts it that way too.

What you see

- **A Voices row** under the two rows of chord names. Each finding is a block over the chord the parts arrive on, named in a few letters: **5ths** and **8ves** for parallel fifths and octaves, **1sns** for unisons, **LT** for a leading tone that does not rise, **7th** for a seventh that does not fall.
- **Magenta lines on the grid** from the note a part leaves to the note it goes to, where a rule is broken, and **a dashed outline** where a leading tone or a seventh should have gone.

- **A line of colour for each part**, from each of its notes to the next, over the notes, so the eye can follow one part through the whole pattern, as on a score. A held note has no line: the part stays where it is. The moves at the change you point at are drawn bolder; pointing anywhere in a chord picks the change that led into it.
- **A line above the grid**. When you point at a change with nothing to mark, it names each part from the top down, with where it goes, then the motion between the outer parts: “Piano: top D5 → C5 · 2nd F4 → E4 · 3rd B3 → G3 · lowest G2 → C3; outer parts in contrary motion”. On one track the parts are named by place; with several tracks, by track. Its tooltip says each move in words (“down a major second from D5 to C5, a step”), and the motion between each pair of parts next to each other. When you point at a step with a finding, the line says what happens there instead, in magenta, and its tooltip says why classical part writing avoids it. With nothing pointed at, it sums the check up: “Voice leading, 4 parts: parallel fifths (1), ...”.

The tooltip spells each interval in the key: B3 to G3 is a major third, not “four semitones”, and F# to G is a minor second. A note struck again shows as held on the line, and the tooltip says “strikes C5 again”, so you can tell it from a note held over.

Which notes make a part

The check needs parts, but the piano roll has tracks. A track that plays one note at a time is one part. A track that plays chords, like a piano, holds several parts. The app splits it the way you would read a score: the top note of each chord goes to the top part, the next note to the next part, and each note joins the part whose last note was nearest. The messages call them “Right hand (upper voice)” and “Right hand (lower voice)”, or “Piano (top voice)”, “Piano (voice 2 of 4)” and “Piano (lowest voice)”.

With **Ghost notes** on, the other tracks are checked too. Drum tracks are left out.

A rest of a quarter note or more ends a line, so the notes on either side of it are not checked against each other. A pair of parts that plays in octaves almost all the time, like double basses doubling the cellos, is one line made louder, not two parts in parallel octaves: at least four notes together, seven in eight of them an octave or a unison apart. The check counts such a pair once, and the summary says “one line doubled in octaves”.

The rules

No parallel fifths or octaves

Take two parts a perfect fifth apart. If both move the same way into another perfect fifth, those are **parallel fifths**. The same goes for octaves and unisons. These intervals blend so well that, for that moment, the two parts sound like one. The texture loses a line, so classical part writing avoids them. A fifth plus an octave (a twelfth) still counts as a fifth.

When the parts move in opposite directions, from a fifth to a twelfth, they make **fifths by contrary motion**. Each part moves differently, but the ear still hears two fifths in a row, so strict part writing avoids those too. Octaves by contrary motion show up now and then between the top part and the bass at a cadence, and the message says so.

C major to D minor with every part rising a step is the textbook case: the soprano G4 and the tenor C4 are a fifth apart, and go to A4 and D4, a fifth apart again: "Parallel fifths: Soprano and Tenor go from G4 over C4 (a fifth) to A4 over D4 (a fifth), both up." The tenor and the bass, C4 over C3 to D4 over D3, make parallel octaves.

A held or repeated note does not count. Both parts have to move.

The leading tone rises

The **leading tone** is the note a semitone below the tonic: B in C major, G# in A minor. When a dominant chord (V, V7, vii°, vii°7) goes to the tonic, the leading tone pulls up to the tonic.

The rule is strict in the top part and the bass, where the ear follows the line. An inner part is freer. In V7 to I, an inner leading tone often drops to the fifth of the tonic chord, so that chord gets all three notes. Textbooks call this a "frustrated" leading tone, and the check leaves inner parts alone.

The check knows three places where the leading tone must rise:

- **V or vii° to I**, the cadence.
- **V to vi** (V to VI in minor), the **deceptive cadence**. The leading tone still rises, to the tonic, which is now the third of vi.
- **A secondary dominant** going to its chord: D7 to G in C major. D7 borrows F# from outside the key, and F# is G's leading tone, so it rises to G. C7 going to F in C major is also a secondary dominant (V7/IV), but its leading tone E belongs to the key, and keeps the freedom it has in C.

If both chords have the leading tone, it may stay where it is. A leading tone going down a scale (C, B, A) is a passing note, not one that has to rise. And V followed by I^6_4 is no arrival, because the parts move on before the leading tone has anywhere to go.

When two parts both have the leading tone, the message also says that it is doubled. Classical part writing never doubles the leading tone: both copies cannot rise without making parallel octaves.

A seventh falls

In a seventh chord such as G7 (G B D F), the seventh, F, is a dissonance. It **resolves down by step** into the next chord. In G7 to C, F falls to E. If the soprano's F5 rises to G5 instead, the check says "Seventh not falling: Soprano has F5, the seventh of G7 (V7), and rises to G5 in C (I) instead of falling a step to E5", and the dashed outline marks E5.

The check reads a few chords the way harmony books do. F A C D going to G is named F6 by its notes, but it works as Dm7 with F in the bass, ii^6_5 in C. Its seventh is C, which should fall to B. A diminished seventh chord's root comes from where it goes: G# B D F going to A has its root on G#, and its seventh is F.

The textbook exceptions are not flagged:

- **The next chord has the note.** Then the seventh may stay where it is.
- **A suspension.** The seventh is held into the next chord and falls a little later.
- **Passing and neighbour notes.** A short note that comes and goes by step makes a seventh only in passing. So does a note added off the beat while the other parts hold theirs.
- **V^4_3 to I^6 .** The seventh (F in G7 with D in the bass) may rise to G while the bass rises from D to E a tenth below it.
- **A passing chord.** In C, then Em7 over D, then C over E, the bass's D is a passing note between two forms of the same C chord.
- **Chromatic voice leading.** A seventh can rise a semitone into the next chord's own leading tone. In Am7 with G in the bass going to E7 with G# in the bass, G rises to G#. The line keeps rising, and the next chord resolves it.
- **The bass in another octave.** The bass may take its note an octave away: B2 to C2 is the leading tone going to the tonic.

What the check does not judge

These are the rules of one style: the part writing of Bach through Brahms, and of harmony exercises. Pop and jazz move whole chords in parallel on purpose, and so did Debussy. Nobody expects a jazz seventh to resolve. Read the check as a teacher of the classical style, not a judge of all music. Where the key is a twelve-tone row, the check follows the rules of twelve-tone writing instead (see [The rules of twelve-tone writing](#)), and where it is a pitch-class set, nothing is checked:

```
pub fn of(key: Scale) -> Rules {  
  match key.kind {  
    ScaleKind::Row(_) => Rules::TwelveTone,  
    ScaleKind::Set(_) => Rules::Neither,  
    _ => Rules::Classical,  
  }  
}
```

Some rules from harmony books are not checked: direct (hidden) fifths and octaves in the outer parts, spacing between the parts, parts crossing, and which note of a chord to double.

In the code

Sharing out the notes of a chord: dynamic programming

`voices_of` turns a track's notes into parts. It walks through the chords in time, cutting the notes where their start changes with `chunk_by`. For each chord, `share_out` decides which part each note goes to. Notes and parts are both sorted from highest to lowest, and a note may only go to a part below the one the note above it took, so the parts never cross. Among all the ways to do that, it picks the one with the smallest total distance in semitones.

Trying every way would take too long for a big chord. **Dynamic programming** solves it with a table instead. `cost[i][j]` is the smallest cost of placing the first `i` notes using only the first `j` parts:

```

for i in 1..=notes {
  for j in 1..=lines {
    let skip_part = cost[i][j - 1];
    let new_part = cost[i - 1][j] + NEW_PART;
    let take = cost[i - 1][j - 1] + distance(i - 1, j - 1);
    cost[i][j] = skip_part.min(new_part).min(take);
  }
}

```

Each cell has three choices. Skip part `j`, so it gets no note in this chord. Start a new part for note `i`, at the large cost `NEW_PART` (10,000, more than any distance), so it happens only when a chord has more notes than there are parts. Or put note `i` on part `j`, at the cost of the distance between them. Each cell is built from cells already filled in, so the whole table takes `notes × parts` steps. Walking back from the last cell recovers which choice made each cost, and so where each note went. The [twelve-tone check](#) uses the same idea to find which forms of a row are played.

Comparing two parts note against note

`together` lists every moment either part starts a note, with the note each part is on at that moment:

```

onsets
  .into_iter()
  .map(|tick| a.current(tick).zip(b.current(tick)))
  .collect()

```

`Option::zip` turns two `Option`s into an `Option` of a pair. It is `Some` only when both are `Some`, that is, when both parts are playing. `consecutive` then looks at each pair of neighbouring moments, and `kind_of_move` decides what the move makes:

```

fn kind_of_move(x: Hop, y: Hop) -> Option<SlipKind> {
  let moves = |(from, to): Hop| {
    let rested = to.start_tick() >= from.end_tick() + LINE_BREAK;
    (from.pitch != to.pitch && !rested).then(|| to.pitch > from.pitch)
  };
  let (up_x, up_y) = (moves(x)?, moves(y)?);
  let first = perfect(x.0.pitch.midi().abs_diff(y.0.pitch.midi()))?;
  let second = perfect(x.1.pitch.midi().abs_diff(y.1.pitch.midi()))?;
  let same_way = up_x == up_y;
  match (first, second) {
    (Perfect::Fifth, Perfect::Fifth) if same_way =>
Some(SlipKind::ParallelFifths),
    (Perfect::Fifth, Perfect::Fifth) =>
Some(SlipKind::ContraryFifths),
    (Perfect::Unison, Perfect::Unison) =>
Some(SlipKind::ParallelUnisons),
    (Perfect::Fifth, _) | (_, Perfect::Fifth) => None,
    _ if same_way => Some(SlipKind::ParallelOctaves),
    _ => Some(SlipKind::ContraryOctaves),
  }
}

```

`moves` gives `Some(true)` for a part that goes up, `Some(false)` for one that goes down, and `None` for a held note or a rest of a quarter note or more: `bool::then` runs its closure only when the condition holds. Each `?` returns `None` from the function as soon as something is missing, and `perfect` gives `None` for an interval that is not a unison, fifth or octave. What is left after the `?` s is the interesting case, two perfect intervals in a row, and a `match` on the pair names it. The arms are tried in order, so the fifth-to-something-else arm must come before the two octave arms that catch everything left.

Naming a part with a closure

A finding knows its parts by number. Only the app knows the tracks' names, so `describe` takes a function that names a part:

```

pub fn describe(&self, voice_name: impl Fn(usize) -> String) -> String {

```

The piano roll passes a closure:

```

let name = |voice: usize| voice_name(drawing.project, check, voice);
// ...
let why: Vec<String> = here.iter().map(|slip|
slip.describe(name)).collect();

```

`name` is passed by value, once for each finding, and yet it can still be used afterwards. That works because the closure only holds shared references (`&Project` , `&VoiceLeading`). A closure whose captures are all `Copy` is itself `Copy` , so each call gets its own copy. (Clippy says so too: it flags `describe(&name)` as a needless borrow.)

Checking only when something changed

Checking a pattern takes under a millisecond in a release build, but the piano roll draws 60 frames a second while a song plays. So it keeps the last check with what it was made from:

```
#[derive(Debug, Clone, PartialEq)]
struct VoiceInput {
    pattern: usize,
    /// The notes of each channel checked.
    notes: Vec<(usize, Vec<NoteEvent>)>,
    time: TimeSignature,
    /// The key on each beat.
    keys: Vec<Scale>,
}
```

Each frame builds a new `VoiceInput` and compares it with the stored one. `#[derive(PartialEq)]` compares field by field, down to every note. Comparing is much cheaper than checking, and when nothing changed the old check is used:

```
if let Some((made_from, check)) = &self.voice_check
    && *made_from == input
{
    return Rc::clone(check);
}
```

`if let ... && ...` is a **let chain**, from the 2024 edition. It tests a pattern and a condition in one `if`.

The check is kept in an `Rc<VoiceLeading>` , a shared pointer. Drawing needs the check while it also changes the piano roll (`&mut self`). Borrowing the check from `self` would lock `self` for the whole frame, so the borrow checker would refuse it.

`Rc::clone` only adds one to a count, so the frame gets a handle of its own for the cost of an integer increment, with no copy of the findings.

Following the parts through every change

`voice_motion::changes` walks through the moments where a new note starts in any part. For each part it takes the note it had just before and the note it has there:

```
fn before(voice: &Voice, tick: usize) -> Option<NoteEvent> {
    let index = voice
        .notes
        .partition_point(|note| note.start_tick() < tick)
        .checked_sub(1)?;
    let note = voice.notes[index];
    (tick < note.end_tick() + LINE_BREAK).then_some(note)
}
```

The notes of a part are sorted by start, so `partition_point` finds with a binary search how many start before `tick`. The one before that count is the part's last note; `checked_sub(1)?` gives up when there is none. After a rest of `LINE_BREAK`, a quarter note, the line has ended, and the next note starts a new one. `bool::then_some` turns the condition into an `Option: Some(note)` when it holds and `None` when it does not.

Naming the motion

```
pub fn motion(upper: &Move, lower: &Move, key: Scale) -> Motion {
    let (a, b) = (upper.semitones(), lower.semitones());
    match (a.signum(), b.signum()) {
        (0, 0) => Motion::Static,
        (0, _) | (_, 0) => Motion::Oblique,
        (x, y) if x != y => Motion::Contrary,
        _ => {
            let number = |upper: Note, lower: Note| {
                let (upper, lower) = (
                    key.spell(upper.pitch_class()),
                    key.spell(lower.pitch_class()),
                );
                lower.letter.steps_up_to(upper.letter)
            };
            if number(upper.from.pitch, lower.from.pitch) ==
                number(upper.to.pitch, lower.to.pitch)
            {
                Motion::Parallel
            } else {
                Motion::Similar
            }
        }
    }
}
```


`signum` is -1, 0 or 1: down, held or up. Matching on the pair of signs sorts out the easy cases first. Both held. One held (the `|` in a pattern means “or”). Opposite signs. Only when both go the same way does the interval matter. Then the closure `number` spells both notes in the key and counts letters from the lower to the upper: E over G is `G.steps_up_to(E)`, 5 letters up, a sixth. Counting letters, not semitones, is what makes a major sixth going to a minor sixth parallel (see [Intervals count letters too](#)).

Keeping the line while you read it

The line above the grid shows the change at the step under the pointer. But to read its tooltip, the pointer has to leave the notes for the line, and then no step is under it. So the piano roll remembers the last step pointed at, and the line uses it while the pointer is on the line itself:

```
let line = Rect::from_min_size(
    ui.cursor().min,
    vec2(ui.available_width(), ui.text_style_height(&TextStyle::Body)),
);
let step = self
    .hovered_step
    .or(drawing.playhead)
    .or(self.last_step.filter(|_| ui.rect_contains_pointer(line)));
```

`ui.cursor()` is where egui will put the next widget, so `line` is the rectangle the line is about to take, worked out before it is drawn. `Option::filter` keeps `last_step` only when the closure says yes: when the pointer is inside that rectangle. While a song plays, the playhead picks the step when the pointer is elsewhere.

Testing it

You learn harmony from this check, so a wrong mark would teach you something wrong. Two kinds of test guard it:

- **Textbook examples.** The tests in `voice_leading.rs` write small four-part examples, about twenty of them, each part as a line of note names. Each test writes a rule and its exceptions as notes and checks what is found: a textbook cadence with nothing to find, parallel fifths by a twelfth, a held note, a rest that ends a line, a doubled line, the leading tone in an outer and an inner part, the deceptive cadence, a secondary dominant, ii^6_5 , V^4_3 to I^6 , a suspension, a passing chord, a note added off the beat, chromatic voice leading, the bass in another octave, a six-four chord, and a track of chords split into parts.

- **Bach's chorales.** The music21 library, used by music scholars, has its own check for parallel fifths and octaves. Both were run on 350 Bach chorales, and they found the same 129 parallels, at the same places between the same parts. For the leading tone and the seventh, every one of the findings left in those chorales is a real departure from strict part writing, such as a soprano leaping away from the leading tone at a cadence. Bach broke the rules now and then, on purpose and with good reason.

```
#[test]
fn a_textbook_cadence_has_nothing_to_avoid() {
    // I IV V7 I in C major, as harmony books write it.
    let check = satb(["C5 C5 B4 C5", "G4 A4 G4 G4", "E4 F4 F4 E4", "C3 F3
G3 C3"]);
    assert_eq!(kinds(&check), vec![]);
    assert_eq!(
        check.summary(),
        "Voice leading, 4 parts: nothing classical part writing avoids."
    );
}
```

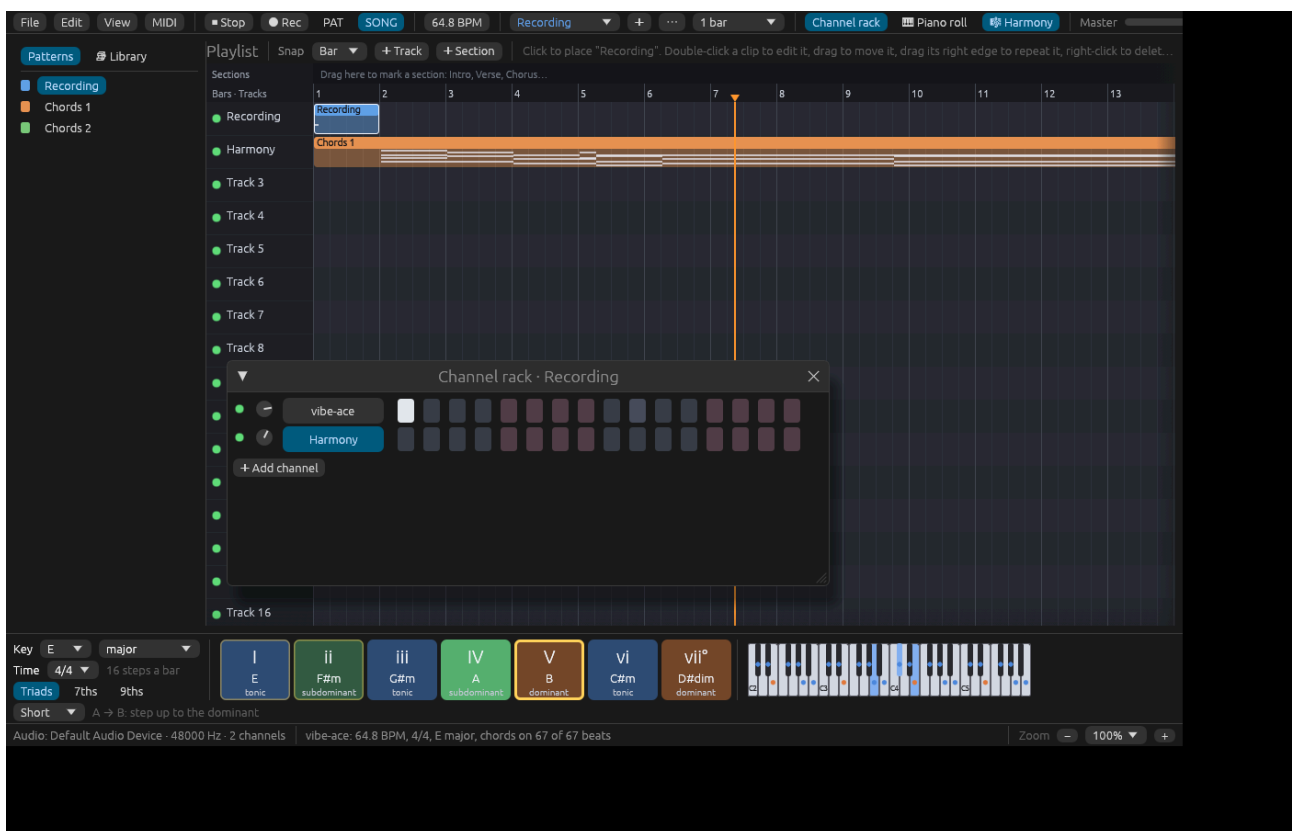
`voice_motion.rs` follows a short chorale on one track, C major (C3 G3 E4 C5) to F major (F2 A3 F4 C5) to G7 (G2 B3 F4 D5), and checks the line of moves at the first change, “S C5 held · A E4 → F4 · T G3 → A3 · B C3 → F2; outer parts in oblique motion”; the sizes held, step, step and leap; the words (“B: down a perfect fifth from C3 to F2, a leap”); the alto and tenor in parallel motion, parallel sixths, and the tenor and bass in contrary motion. At the second change the soprano and the bass both rise a step, a fifth and two octaves apart: parallel motion, and parallel fifths. Other tests check a melody over a held note (oblique), and that a fifth to a twelfth, or a major sixth to a minor sixth, is still parallel.

Exercises on voice leading are in [Exercises: music theory in code](#).

Importing a recording (experimental)

Files: `crates/lineup-listen/` and `crates/lineup-app/src/app/import.rs`

File > Import recording..., or `ferrous-lineup song.mp3` on the command line, opens an MP3, WAV, FLAC or Ogg file as a new project, and tries to play along with it. The recording itself goes on a sampler channel, and a **Harmony** channel plays the chord heard on every beat. The tempo, the time signature, the key and the key changes are set from what was heard, so the bar lines fall on the recording's bars, the chord sheet fills in, and the harmony keys light up with the chord that is playing. How to use it is in the manual's [Recordings](#).



It is experimental, and the chapter title says so: the tempo, the chords, the key changes and the first downbeat are estimates, not a score read from the sound. A file of notes, MIDI, says exactly what is played; a recording is a single wave, every instrument added together, and everything has to be worked out from it.

The work is a crate of its own, `lineup-listen`, which knows nothing of windows, plus one file in the app. This chapter follows the sound from the file to the project: a Fourier transform to see the notes, and **dynamic programming** three times, for the beats, the chords and the key changes.

The pipeline

`analyze` is the whole of the listening, as plain functions, each in a module of its own, each taking the data the one before it made:

```

pub fn analyze(recording: &Recording) -> Analysis {
    let beats = beats::track(&recording.samples,
recording.sample_rate).times;
    let bpm = tempo_of(&beats);

    let whole = bpm_key::analyze(
        &recording.samples,
        1,
        recording.sample_rate,
        &bpm_key::Config::default(),
    )
    .ok()
    .and_then(|features| features.key)
    .map(|estimate| {
        let tonic = PitchClass::from_semitone(estimate.key.tonic.index());
        let kind = if estimate.key.is_minor() {
            ScaleKind::NaturalMinor
        } else {
            ScaleKind::Major
        };
        Scale::new(tonic, kind)
    });

    let frames = chroma::frames(recording);
    let beat_chromas: Vec<_> = beats
        .iter()
        .enumerate()
        .map(|(i, &start)| {
            let end = beats.get(i + 1).copied().unwrap_or(start + 60.0 /
bpm);
            chroma::average(&frames, start, end)
        })
        .collect();
    let chords = chords::chords_per_beat(&beat_chromas, whole);
    let (beats_per_bar, first_downbeat) = meter::guess(&chords);
    let keys = keys::key_spans(&beat_chromas, beats_per_bar,
first_downbeat);
    let scale = keys::main_key(&keys).or(whole);

    Analysis {
        beats,
        bpm,
        beats_per_bar,
        first_downbeat,
        scale,
        keys,
        chords,
    }
}

```

In order:

1. **Beats** (`beats.rs`): when each beat falls, in seconds, and from them the tempo.

2. **Key** (the `bpm-key` crate): the key of the whole recording, which helps name the chords.
3. **Chroma** (`chroma.rs`): how strongly each of the twelve pitch classes sounds, frame by frame, then averaged over each beat.
4. **Chords** (`chords.rs`): the chord of each beat.
5. **Meter** (`meter.rs`): three or four beats to the bar, and which beat is the first downbeat.
6. **Key changes** (`keys.rs`): the key of each bar, found by the same code that finds the key of a song's notes. The key of the longest stretch in one key, `keys::main_key`, becomes the project's key; `or(whole)` keeps the whole recording's key for a recording where nothing sounds long enough to have one.

Then `arrange.rs` turns the `Analysis` into a `Project`, with the types the rest of Ferrous Lineup uses: `Channel`, `Pattern`, `NoteEvent`, `Clip`. Nothing after that knows the notes came from a recording.

The chain after `bpm_key::analyze` reads the way it runs. The crate returns a `Result`; `.ok()` turns it into an `Option`, dropping the error, since a recording without a clear key is not a failure. `and_then` goes on only if there is a value, and the features may have no key (`features.key` is itself an `Option`). `map` turns the crate's key into one of ours. If any step has nothing, `whole` is `None`, with no `if` and no `match`. The last beat has no next beat to end at, so `unwrap_or` gives it one beat's length.

Decoding with Symphonia

An audio file is a **container** (WAV, MP3, Ogg...) holding one or more **tracks**, each compressed with a **codec**. The [Symphonia](#) crate, written in pure Rust, so there is no C library to install on any system, separates the two: you *probe* the container, pick its audio track, and give the track's **packets** to a decoder for its codec. Each packet decodes to a block of samples.

Symphonia reads from anything that implements its `MediaSource` trait: a file, or a buffer in memory. It takes it as a `Box<dyn MediaSource>`:

```
let file = File::open(path).map_err(DecodeError::Io)?;
let stream = MediaSourceStream::new(Box::new(file),
MediaSourceStreamOptions::default());
```

`dyn MediaSource` is a **trait object**: "some type that implements `MediaSource`", with the method to call looked up while the program runs. Different types have different sizes, so a trait object always lives behind a pointer, here a `Box`. `Box::new(file)` turns the `File` into one without a word more.

The decoder loop shows how errors are *matched*, not just passed on with `?`:

```
loop {
  let packet = match format.next_packet() {
    Ok(Some(packet)) => packet,
    Ok(None) => break,
    // A file cut short ends with an I/O error: keep what we have.
    Err(SymphoniaError::IoError(_)) => break,
    Err(err) => return Err(format_error(err)),
  };
  if packet.track_id != track_id {
    continue;
  }
  let block = match decoder.decode(&packet) {
    Ok(block) => block,
    // One damaged packet is a click, not a reason to give up.
    Err(SymphoniaError::DecodeError(_)) => continue,
    Err(err) => return Err(format_error(err)),
  };
  let channels = block.spec().channels().count().max(1);
  sample_rate = block.spec().rate();
  block.copy_to_vec_interleaved(&mut interleaved);
  samples.extend(
    interleaved
      .chunks(channels)
      .map(|frame| frame.iter().sum::<f32>() / channels as f32),
  );
  if too_long(samples.len(), sample_rate) {
    return Err(DecodeError::TooLong);
  }
}
```

An I/O error at the end means a file cut short, so the loop keeps what it has; a decode error means one damaged packet, so it skips it; any other error ends the import.

`format_error` is a closure defined just before, `|err: SymphoniaError| DecodeError::Format(err.to_string())`, which turns Symphonia's error into ours.

Samples come **interleaved**: left, right, left, right. `chunks(channels)` cuts the list into one slice per moment, and each slice's average is one sample of a mono recording: chords do not care about left and right. A recording longer than `MAX_SECONDS`, ten minutes, is refused while it decodes, since the analysis holds the recording, and a copy of it, in memory.

Symphonia's codecs are Cargo **features**, turned on in the workspace `Cargo.toml`, so that a build only carries the ones it uses:

```
symphonia = { version = "0.6", default-features = false, features =
  ["mp3", "wav", "pcm", "flac", "ogg", "vorbis"] }
```

From samples to chroma

A chord is a set of pitch classes: C major is C, E and G in any octave (see [Pitch classes, notes and spelling](#)). To find chords in sound, we need to know, moment by moment, how strongly each of the twelve pitch classes sounds. Those twelve numbers are called the **chroma**, and they come from a **Fourier transform**, which takes a stretch of samples and says how much of each frequency is in it.

`chroma::frames` cuts the recording into overlapping **frames** of 8,192 samples, about 0.19 s each at 44.1 kHz, starting a new one every 2,048 samples (46 ms). Each frame is multiplied by a **Hann window**, a smooth bump that fades the frame in and out: without it, the sudden cut at each end would look to the transform like a click, which contains every frequency. The FFT (the fast Fourier transform, from the `realfft` crate) then says how loud each frequency **bin** is in the frame. Bin `k` is the frequency `k × sample_rate / 8192`, so at 44.1 kHz the bins are 5.4 Hz apart. Neighbouring notes are further apart than that from about G2 (98 Hz) up; the frame is long because low notes need it.

Each bin belongs to a note. The formula `69 + 12 × log2(f / 440)` turns a frequency into a MIDI note number: 69 is A4, at 440 Hz, and every semitone up multiplies the frequency by the twelfth root of two. Rounding gives the nearest note, and the remainder by 12 its pitch class:

```
let pitch_class_of = |bin: usize| {
    let frequency = bin as f32 * rate / FRAME as f32;
    let midi = 69.0 + 12.0 * (frequency / 440.0).log2();
    (midi.round() as i32).rem_euclid(12) as usize
};
```

`rem_euclid` is the remainder that is never negative: `-3 % 12` is `-3` in Rust, `(-3).rem_euclid(12)` is `9`. No bin is below MIDI note 0 here, but the closure does not have to know that.

Every bin's loudness is added to its pitch class. Three details make this work on real music:

- The loudness is taken as `ln(1 + 100 × magnitude)`, because we hear loudness on a roughly logarithmic scale: a quiet note an octave up should still count, and the loudest note should not drown everything.

```
let level = |bin: usize| (1.0 + 100.0 * spectrum[bin].norm()).ln();
```

- Higher notes are further apart in Hz, so they get more bins than low ones. Each pitch class is divided by its number of bins, or B would always look louder than C.

- Two chromas are kept per frame. The **treble** one, from 130 to 2,100 Hz (about C3 to C7), is where the chord is. The **bass** one, from 40 to 260 Hz (about E1 to C4), mostly says which note is lowest, and the lowest note is usually the chord's root.

Which pitch class each bin adds to is worked out once, before the loop over the frames, as a list of `(bin, pitch class)` pairs:

```
let bins = |(low, high): (f32, f32)| {
  let first = (low * FRAME as f32 / rate).ceil() as usize;
  let last = ((high * FRAME as f32 / rate).floor() as usize).min(FRAME / 2);
  (first.max(1)..=last)
    .map(|bin| (bin, pitch_class_of(bin)))
    .collect::<Vec<_>>();
};
let treble_bins = bins(TREBLE);
let bass_bins = bins(BASS);
```

The closure's parameter is a **pattern**: `|(low, high): (f32, f32)|` takes a tuple and names its two halves, as `let (low, high) = range;` would. `TREBLE` and `BASS` are such tuples. With the table made, the loop over thousands of frames does nothing but additions.

The FFT itself is planned once too. `RealFftPlanner` picks the fastest way to transform 8,192 real numbers and gives it back as an `Arc<dyn RealToComplex<f32>>`, another trait object, which the loop runs on every frame with buffers made once:

```
let mut planner = RealFftPlanner::<f32>::new();
let fft: Arc<dyn RealToComplex<f32>> = planner.plan_fft_forward(FRAME);
let mut input = fft.make_input_vec();
let mut spectrum = fft.make_output_vec();
let mut scratch = fft.make_scratch_vec();
```

`chroma::average` then gives one chroma per beat: the average of the frames whose middle falls between this beat and the next.

From chroma to chords: templates and Viterbi

`chords::chords_per_beat` names the chord of each beat. There are two ideas, the classic way of doing this without machine learning.

Templates. Every chord we can recognize is a **template**: twelve numbers, 1 for its notes and 0 for the rest. There are six qualities on twelve roots, 72 chords, plus “no chord”:

```
const QUALITIES: [ChordQuality; 6] = [
    ChordQuality::Major,
    ChordQuality::Minor,
    ChordQuality::Dominant7,
    ChordQuality::Minor7,
    ChordQuality::Major7,
    ChordQuality::Diminished,
];
```

The beat's chroma is compared with each template by **correlation**: both are moved to an average of 0 and a length of 1, and their dot product is then 1 if they rise and fall together and 0 if unrelated. It is the same comparison the key finder makes with its key profiles (see [Hearing harmony](#)). Then three small adjustments:

```
let Some(chord) = candidate.chord else {
    return NO_CHORD_SCORE;
};
let mut score = dot(&treble, &normalized(&candidate.template));
score += BASS_WEIGHT * bass[usize::from(chord.root.semitone())] /
bass_peak;
if scale.is_some_and(|scale|
chord.pitch_set().is_subset_of(scale.pitch_set())) {
    score += KEY_WEIGHT;
}
if chord.quality.intervals().len() > 3 {
    score -= SEVENTH_COST;
}
score
```

- A strong bass note on the chord's root adds up to 0.15 (`BASS_WEIGHT`), and a weak one takes a little away, since the bass chroma is centred on its average too.
- A chord whose notes all belong to the key adds 0.08 (`KEY_WEIGHT`). Most chords of a song do, so the key is a useful **prior**: what we expect before listening.
- A four-note chord loses 0.03 (`SEVENTH_COST`), so a plain triad wins unless the seventh is really there. C7 contains the whole C major chord, and without this it would win whenever a B $\flat$ is faintly in the air.

"No chord" always scores 0.25 (`NO_CHORD_SCORE`): a beat where no chord does better, such as silence, drums alone or speech, has none.

Smoothing. Taken alone, the best chord of each beat jumps around: a passing melody note makes C look like Am for one beat. Real chords last. So changing chord costs 0.25 (`CHANGE_COST`), and the **Viterbi algorithm** finds the sequence of chords with the best total score. It is dynamic programming: for each beat and each of the 73 candidates, it keeps the best score of any sequence that ends there, made from the previous beat's bests, and remembers which chord came before:

```

let mut best = scores(&beats[0], &candidates, scale);
let mut from = vec![vec![0_usize; count]; beats.len()];
for (b, beat) in beats.iter().enumerate().skip(1) {
    let here = scores(beat, &candidates, scale);
    // Changing chord costs the same from anywhere, so the best way to
    // change into any chord is from the overall best one.
    let (leader, leader_score) = best
        .iter()
        .copied()
        .enumerate()
        .fold((0, f32::MIN), |a, (i, s)| if s > a.1 { (i, s) } else { a
});
    let mut next = vec![0.0; count];
    for c in 0..count {
        let (previous, total) = if best[c] >= leader_score - CHANGE_COST {
            (c, best[c])
        } else {
            (leader, leader_score - CHANGE_COST)
        };
        from[b][c] = previous;
        next[c] = total + here[c];
    }
    best = next;
}

```

Each chord at beat `b` is reached either by staying on it, or by changing from some chord and paying the penalty. Comparing every pair would be 73×73 comparisons per beat. But changing costs the same from anywhere, so the best change into any chord comes from the previous beat's overall best, the `leader`: one comparison per chord, 73 per beat. At the end, the code takes the best last chord and walks back through `from` to read off the whole sequence. The key finder uses the same loop over bars and keys (see [Where the key changes](#)).

Finding the leader needs the largest `f32`, and `f32` has no `max` for an iterator: floats are not `Ord`, because `NaN` compares with nothing. `fold` does it by hand, carrying the best `(index, score)` so far. The beat tracker below does the same with `max_by` and `total_cmp`, a comparison that puts every float, `NaN` too, in some order.

Beats: onsets, autocorrelation and dynamic programming

`beats::track` is the beat tracker of D. P. W. Ellis, “Beat Tracking by Dynamic Programming” (2007), with the settings the `librosa` library documents as its defaults, so results can be checked against it. It has three stages.

Onset strength. How much *new* sound starts in each short frame: 2,048 samples, a new one every 512 (12 ms at 44.1 kHz). A spectrogram on the **mel scale**, 128 bands

spaced the way the ear hears them, narrow at the bottom and wide at the top, is taken in decibels, and for each frame the increases of every band since the previous frame are added up. Decreases do not count: a note dying away is not an onset. The result, one number per frame, is the **onset envelope**, which peaks on every drum hit and note start.

Tempo. The envelope is compared with itself shifted by every possible beat length, from 30 to 320 BPM: its **autocorrelation**. At the right shift, peaks land on peaks and the correlation is high. Shifts near 120 BPM get a head start, a bell curve one octave wide, because trackers otherwise pick half or double the tempo as readily as the tempo itself:

```
let bpm = frame_rate * 60.0 / lag as f64;
let octaves = (bpm / LIKELY_BPM).log2();
let pull = (-0.5 * (octaves / LIKELY_SPREAD_OCTAVES).powi(2)).exp();
let score = correlation(lag) * pull;
```

At 60 or 240 BPM, an octave away, a correlation counts 0.61 times as much as the same correlation at 120.

Beats. Of all the ways to place beats, the tracker wants the one where beats land on strong onsets and are spaced close to the beat length. Trying every placement is impossible, but dynamic programming makes it cheap. The best score for a chain of beats ending at frame `i` is the onset strength there, plus the best score of a chain ending one beat length earlier, give or take, minus a penalty for the give or take:

```

for i in 1..envelope.len() {
    // The previous beat is between half a beat and two beats back.
    let earliest = i.saturating_sub(period * 2);
    let latest = i.saturating_sub((period / 2).max(1)).min(i - 1);
    let mut best: Option<(usize, f64)> = None;
    for (tau, &earlier) in score.iter().enumerate().take(latest +
1).skip(earliest) {
        let spacing = (i - tau) as f64 / period as f64;
        let candidate = earlier - TIGHTNESS * spacing.ln().powi(2);
        if best.is_none_or(|(_, b)| candidate > b) {
            best = Some((tau, candidate));
        }
    }
    match best {
        // A chain worth continuing: positive, so silence does not
        // chain beats that are not there.
        Some((tau, total)) if total > 0.0 => {
            score[i] = strength[i] + total;
            from[i] = Some(tau);
        }
        _ => score[i] = strength[i],
    }
}

```

`spacing` is 1 for a gap of exactly one beat, and its logarithm is then 0: no penalty. Half a beat or two beats give $\ln 2$ squared, about 0.48, times `TIGHTNESS` (100): a beat in the wrong place has to be much stronger to win. `saturating_sub` subtracts but stops at 0 instead of wrapping around, which unsigned numbers would otherwise do near the start. `is_none_or` is true for `None` or when the closure says so: “there is no best yet, or this one is better”, in one call. Each frame remembers in `from` which earlier beat it chose, and at the end the best last frame is traced back to the start:

```

let mut at = (0..envelope.len())
    .max_by(|&a, &b| score[a].total_cmp(&score[b]))
    .unwrap_or(0);

```

The autocorrelation only knows whole frames. At 44.1 kHz, 86 frames a second, the lags around 120 BPM are 42, 43 and 44 frames: 123.0, 120.2 and 117.5 BPM, nothing in between. So the tempo the project gets is not the tracker’s but the slope of the beats themselves, the straight line closest to “the time of beat n ”, a **least-squares fit**:

```

fn tempo_of(beats: &[f64]) -> f64 {
    let n = beats.len() as f64;
    if beats.len() < 2 {
        return 120.0;
    }
    let mean_index = (n - 1.0) / 2.0;
    let mean_time = beats.iter().sum::<f64>() / n;
    let (mut covariance, mut variance) = (0.0, 0.0);
    for (i, &time) in beats.iter().enumerate() {
        let di = i as f64 - mean_index;
        covariance += di * (time - mean_time);
        variance += di * di;
    }
    let seconds_per_beat = covariance / variance;
    if seconds_per_beat > 0.0 {
        60.0 / seconds_per_beat
    } else {
        120.0
    }
}

```

The slope of the best line is the covariance of index and time divided by the variance of the index: seconds per beat, averaged over the whole song, with the small wobbles of each beat cancelling out.

Meter: where the bars are

Chords mostly change on the first beat of a bar. So `meter::guess` counts, for each way of grouping the beats (in threes or fours, starting on any beat), how many chord changes land on a group's first beat, and keeps the grouping that beats chance by the most:

```

pub fn guess(chords: &[Option<Chord>]) -> (usize, usize) {
    let changes: Vec<usize> = (1..chords.len())
        .filter(|&i| chords[i] != chords[i - 1] && chords[i].is_some())
        .collect();
    if changes.len() < 4 {
        return (4, 0);
    }
    let mut best = (4, 0, f64::MIN);
    for beats in [4, 3] {
        for phase in 0..beats {
            let on_phase = changes.iter().filter(|&i| i % beats ==
phase).count();
            let mut score = on_phase as f64 / changes.len() as f64 - 1.0 /
beats as f64;
            if beats == 4 {
                score += FOUR_PRIOR;
            }
            if score > best.2 {
                best = (beats, phase, score);
            }
        }
    }
    (best.0, best.1)
}

```

If chords changed at random, a quarter of the changes would land on any one beat of four, and a third on any one beat of three. Subtracting that chance, `1.0 / beats`, makes the two meters comparable: without it, grouping in threes would always look better, since each group's first beat catches a larger share. A small head start, `FOUR_PRIOR` (0.05), goes to 4/4, since most popular music is in four. With fewer than four changes there is nothing to go on, and the answer is 4/4 from the first beat. The result is `(beats per bar, first downbeat)`, and the downbeat is a beat index: `phase`, the first beat in the recording where a bar starts.

Key changes

`bpm-key` gives one key for the whole recording. For where the key changes, `keys.rs` hands the work to `lineup_core::harmony::key_finding`, the code that finds key changes in a song's notes (see [Where the key changes](#)). It asks for a chroma per bar. A recording has no notes, but it has the chroma of every beat, so the beats are put together into bars, and each bar's chroma stands for the notes in it:

```

fn bar_chroma(beats: &[Frame]) -> key_finding::Chroma {
    let mut sum = [0.0; 12];
    for beat in beats {
        let floor = beat.treble.iter().copied().fold(f32::MAX, f32::min);
        let chroma = beat.treble.map(|v| v - floor);
        for (pc, s) in sum.iter_mut().enumerate() {
            // The note a fifth below `pc` is five semitones above it.
            *s += (chroma[pc] - FIFTH_OVERTONE * chroma[(pc + 5) %
12]).max(0.0);
        }
    }
    sum
}

```

Two corrections make a sound's chroma look more like a score's. First, in a logarithmic spectrum every pitch class has some energy, from noise and from the overtones of other notes; taking away each beat's quietest pitch class, its **floor**, leaves the notes that stand out.

Second, the overtones themselves. A note played on an instrument is not one frequency but many: the note, twice it, three times it, and so on. Three times the frequency is an octave and a fifth above (a ratio of 3 is 1,902 cents, a hair over 19 semitones). So a recording of C major is full of G (from C), D (from G) and B (from E) that nobody played, and the key comes out a fifth too high, G for C, more often than not. Each pitch class therefore loses 0.3 (`FIFTH_OVERTONE`) of the pitch class a fifth below it. A fifth below is seven semitones down, which in a circle of twelve is five up: `(pc + 5) % 12` . The comment says so, because that line is easy to get wrong.

Only the treble chroma counts here. The low notes' own overtones fill the bass chroma with fifths, and with it the key was right less often: 80% of beats against 84%, as the code's comment records.

`key_spans` cuts the beats into bars with the meter just found, the beats before the first downbeat making a bar of their own, and asks `key_finding` for its spans, counted in bars. Each span is then moved back to beats:


```

let bars: Vec<_> = edges
    .windows(2)
    .map(|bar| bar_chroma(&beats[bar[0]..bar[1]]))
    .collect();
key_finding::key_spans(&bars)
    .into_iter()
    .map(|span| KeySpan {
        start: edges[span.start],
        end: edges[span.end],
        ..span
    })
    .collect()

```

`edges` holds the beat where each bar starts, and one past the last beat. `windows(2)` walks it two at a time, overlapping, `[0, 2]`, `[2, 6]`, `[6, 10]` ..., which is each bar's first beat and the next bar's. `KeySpan { start, end, ..span }` is **struct update syntax**: a new `KeySpan` with these two fields, and every other field (the keys ranked, how sure) taken from `span`.

The song's key, the one the project gets, is the key of the longest of these stretches:

```

pub fn main_key(spans: &[KeySpan]) -> Option<Scale> {
    spans
        .iter()
        .max_by_key(|span| span.end - span.start)
        .and_then(KeySpan::key)
}

```

`max_by_key` gives the item with the biggest key, here the most beats. It is right more often than `bpm-key`'s single key (see [How well it works](#)): a song that modulates, or opens in another key, pulls the notes of the whole recording away from its home key, and the home key is usually where the song spends longest.

Arranging

`arrange.rs` builds the project. The recording becomes a sampler channel with one note, at the first step of the song. The sampler plays its file from the start at its own speed, so the recording's first downbeat must fall on a bar line of the song. It rarely does, so the recording is saved again with silence in front, the **lead-in**: just enough for its first downbeat to land on the start of the next whole bar.

```
pub fn lead_in(analysis: &Analysis) -> LeadIn {
    let downbeat = analysis
        .beats
        .get(analysis.first_downbeat)
        .copied()
        .unwrap_or(0.0)
        .max(0.0);
    let bar = analysis.beat_seconds() * analysis.beats_per_bar as f64;
    let bars = (downbeat / bar).ceil() as usize;
    LeadIn {
        bars,
        seconds: bars as f64 * bar - downbeat,
    }
}
```

At 120 BPM in 4/4 a bar lasts 2 s. A first downbeat at 1.3 s moves to 2 s, the start of bar 2: one bar, and 0.7 s of silence. `write_wav` saves the copy, as 32-bit float samples with the `hound` crate, next to the original as `song.ferrous-lineup.wav`. Any beats before the downbeat land in that first bar, as a pickup does in a score.

The chords become **spans**, one chord held over consecutive beats, in song beats. A beat of the recording is moved by the lead-in and the downbeat; one that would land before the song starts is left out:

```
for (beat, chord) in analysis.chords.iter().enumerate() {
    let Some(song_beat) = (beat +
offset).checked_sub(analysis.first_downbeat) else {
        continue;
    };
    let Some(chord) = *chord else {
        continue;
    };
    match spans.last_mut() {
        Some(last) if last.chord == chord && last.first_beat + last.beats
== song_beat => {
            last.beats += 1;
        }
        _ => spans.push(Span {
            chord,
            first_beat: song_beat,
            beats: 1,
        }),
    }
}
```

`checked_sub` returns `None` where a plain `-` on unsigned numbers would underflow, and `let ... else` skips that beat. The `match` on `spans.last_mut()` grows the last span when the same chord goes on from the very next beat, and starts a new span

otherwise. A beat with no chord ends a span, since the next chord is then not right after it.

`follow` makes the project. It sets the tempo, the time signature and the key, adds the sampler channel, named after the file, and a Lineup Osc channel called “Harmony”, and voices each span with the same `library::voicing::voice` the band uses, so each chord moves as little as possible from the last (see [How the band plays a chord sheet](#)). The chords go into patterns of 16 bars, “Chords 1”, “Chords 2”..., laid one after another on a track called “Harmony”; the recording’s pattern is on a track called “Recording”. A span that crosses into the next pattern is cut in two:

```
// A span that crosses into the next pattern is cut in two.
while start < end {
  let index = start / pattern_steps;
  let pattern_end = (index + 1) * pattern_steps;
  let length = end.min(pattern_end) - start;
  if notes_by_pattern.len() <= index {
    notes_by_pattern.resize(index + 1, Vec::new());
  }
  for &pitch in &voicing {
    notes_by_pattern[index].push(NoteEvent {
      velocity: CHORD_VELOCITY,
      ..NoteEvent::new(pitch, start - index * pattern_steps, length)
    });
  }
  start += length;
}
```

The chords are played at velocity 80, softer than the default, to stay behind the recording. Last, `arrange::key_spans` moves the key spans from beats to song steps the same way, so the app can show them and mark them on the playlist.

The import in the app

Listening to a song takes a few seconds, so `app/import.rs` runs it on a thread of its own, the way the WAV export does (see [Knowing when it is done](#)). The thread’s closure returns everything the window needs:

```
type Outcome = Result<(Project, String, Vec<KeySpan>), String>;
```

The project, a summary of what was heard for the status bar, and the key changes in song steps; or why it failed. `type` gives the long type a short name. The summary is a sentence such as “song: 98.2 BPM, 4/4, E minor, chords on 310 of 312 beats”, and it is

written on the worker's thread, so that thread first takes the window's language and note names, which are [settings that live on a thread](#):

```
let language = crate::i18n::Language::current();
let naming = lineup_core::NoteNaming::current();
self.import = Some(Import {
    worker: thread::spawn(move || {
        language.set();
        naming.set();
        listen(&path)
    }),
});
```

`listen` decodes, analyzes, writes the WAV copy (into the system's temporary folder if the original's folder cannot be written to) and builds the project. Every frame, `check_import` asks the handle `is_finished()`; when it is, it `join`s the thread for its result, swaps the project in with the same `replace_project` that File > Open uses, gives the key changes to  **Find the key** and starts playing the song:

```
match import.worker.join() {
    Ok(Ok((project, summary, keys))) => {
        self.replace_project(project, None);
        self.key_finder.new_project(&self.project, keys);
        self.notice = Some(summary);
        self.set_mode(PlayMode::Song);
    }
}
```

The window never waits, so the music keeps playing while a new recording is listened to. `replace_project` also fills the chord sheet with the chords heard in the Harmony track's notes, as it does for any project that comes without chords (see [Writing it down](#)).

Opening another project while an import is listening drops the import:

`replace_project` takes it out of `self.import`, and the status bar says **Stopped importing the recording**. Dropping a `JoinHandle` does not stop its thread; it **detaches** it. The thread listens on to the end, and its result is thrown away, which is what is wanted: the new project stays, and nothing has to interrupt the analysis halfway.

From the menu, the command line or a drop

A recording reaches the import from the menu, from the command line or dropped on the window. A file named on the command line goes through `open_now`, which

opens projects too, and which asks `lineup_listen::is_audio_file` first: an extension in `AUDIO_EXTENSIONS`, `["mp3", "wav", "flac", "ogg"]`, in any case.

A dropped file is harder to place, because a sound dropped on the window can also be a sample for the channel rack (see [The sampler](#)): a WAV of a drum hit is a sound, a WAV of a whole song is a recording. `sample::is_recording` decides by what the sampler can play and for how long:

```
pub fn is_recording(path: &Path) -> bool {
    let is_wav = path
        .extension()
        .and_then(|extension| extension.to_str())
        .is_some_and(|extension| extension.eq_ignore_ascii_case("wav"));
    if !is_wav {
        return true;
    }
    hound::WavReader::open(path).is_ok_and(|reader| {
        let rate = u64::from(reader.spec().sample_rate);
        u64::from(reader.duration()) > rate *
crate::instrument::LONG_SAMPLE_SECONDS
    })
}
```

The sampler plays only WAV, so an MP3, FLAC or Ogg file can only be a recording. A WAV is one when it lasts longer than ten seconds (`LONG_SAMPLE_SECONDS`), the same line past which a sampler voice fades out when playback stops: past it, a sound is treated as a piece of music rather than a hit. `reader.duration()` is the number of samples per channel, read from the file's header without decoding it.

`is_project_file`, which tells the window and the channel rack which dropped files are not sounds, counts a recording in.

Why these crates

Each job went to whatever did it best on the same test recordings: library demos rendered to WAV, where the tempo, the key and the chord of every beat are known, and a few Creative Commons recordings with librosa's results as the reference.

- **Decoding:** Symphonia, in pure Rust, for the four formats at once.
- **Beats:** our own `beats` module. The `beat-track-rs` crate implements the same tracker, and the module gives the same beats on every test recording, about ten times faster. The crate also decodes files, so it would bring a second copy of Symphonia and the `ndarray` crate into every build, for 250 lines of tracker.

- **Key:** `bpm-key` , with the Krumhansl-Kessler profiles, was right most often, and is fast.
- **Chords:** no crate did well enough, so `chroma.rs` and `chords.rs` are our own, about 300 lines.
- **Meter:** nothing available guesses it, hence `meter.rs` .

Neural beat trackers, such as `beat-this` and `beatnet-rs` , were left aside: they need ONNX models, so the import would no longer be pure Rust while it runs.

How well it works

The crate's [README](#) has the tables, measured on 29 library demos rendered to WAV:

Tempo	right on 20, half or double on 8, wrong on 1
Beats on the grid of the true tempo	27 of 29
Key of the whole song (<code>bpm-key</code>)	24 of 29
Chords: right root and major or minor	79% of 3,887 beats
Meter	3/4 on all 5 demos in three, 4/4 on the other 24
First downbeat on a bar line	22 of 29

Half or double tempo is the classic confusion of every tempo tracker, and for playing along it matters little: the beats are still on the beat. Whole songs of plain triads come out right; the misses are seventh chords heard as their triad or the other way round, melodies played in block chords, whose passing notes look like chords of their own, and the diminished chords of the jazzier styles.

For key changes there is a report, an ignored test that renders song demos to audio, every one with key changes and one in five of the others, and compares the keys found with the key markers the demos have from their scores:

```
cargo test --release -p lineup-listen --test key_changes -- --ignored --nocapture
```

Measured on 30 October 2026, on 136 songs, counting a key as right when its tonic and its major or minor are:

Keys, from a recording	
Beats in the right key	73.7% of 36,306
The song's key (its longest stretch in one key)	91 of 136
Key changes in the scores found within two beats	63 of 315
Songs in one key where a change is found anyway	3 of 52

From a song's notes, the key finder gets 87% of bars right (see [How well it hears](#)); from the sound of the same songs, about three beats in four. Changes are seldom invented, but seldom found on the beat either: the chroma of a sound is blurrier than notes, and a new key has to hold long enough to pay for its change. Many are found a bar or two late, as in *Maple Leaf Rag*, where the move to D $\flat$ major at 120 s is heard at 123 s. The report lists every song with its keys and where they change, true and found: the place to look next. On the same songs, the one key `bpm-key` hears in the whole recording was right for 82, which is why the project takes the longest stretch's key instead.

Rough edges

- **A copy of the recording.** The WAV copy with its lead-in exists only because a sampler note plays its sound from the first sample. The sampler can start a long sound partway through (see [Starting partway](#)), so an offset saved with the channel could take the copy's place.
- **Stopping.** A sampler plays its whole sound whatever happens, which is right for a drum hit and wrong for a song, so voices of a sound longer than ten seconds fade out when playback stops (see [The sampler](#)). Ten seconds is a rule of thumb; a setting per channel would be cleaner.
- **Meters.** Only 3/4 and 4/4 are guessed; music in 6/8 comes out as 3/4 at twice the tempo.
- **The whole mix.** The chords come from everything that sounds. A tune in single notes is not followed by the harmony it implies, and a loud melody can pull the chord towards its own notes.

Testing it

Each module has unit tests on data made in the test. The beat tracker gets twenty seconds of clicks at 120 BPM and must find the tempo within 3 BPM, and every beat within 0.06 s of a click; silence must give no chain of beats at all; and the mel filters

must cover the spectrum, overlapping by half. `arrange.rs` builds an `Analysis` by hand, a beat every half second from 0.3 s, the downbeat on the third, and checks the lead-in, the spans and the project:

```
#[test]
fn chords_join_into_spans_in_song_beats() {
    let analysis = analysis();
    let spans = spans(&analysis, lead_in(&analysis));
    let firsts: Vec<_> = spans.iter().map(|s| (s.first_beat,
s.beats)).collect();
    // The two beats before the downbeat land in the lead-in bar.
    assert_eq!(firsts, [(2, 2), (4, 4), (8, 4), (13, 1)]);
}
```

The chords are G G C C C C G G G G, no chord, C. The two G beats before the downbeat land on the last two beats of the lead-in bar (beats are counted from 0); the C at the end is a span of its own, after the beat with no chord. `decode.rs` checks that a recording one sample longer than ten minutes is refused, with the message the user sees; and the app checks that opening another project lets go of an import that is still listening.

Exercises on the import are in [Exercises: music theory in code](#).

Exercises: music theory in code

These exercises practise this part on the real code: notes and chords, numerals, the editing tools, the hearing of chords and keys, sets, rows and micromodes, voice leading and the import of recordings. Each one says what is **already there**, so you extend it rather than build it again. Some are things to play in the app and listen to; some are pieces of Rust with a clear end; the ones called “Think about it” are questions, with no code to write. Many are music theory first: work out the answer on paper, then let the code check it.

Try the Rust ones on a branch of your own. Run `cargo test` after each change, and `cargo fmt --all --check` and `cargo clippy --workspace --all-targets -- -D warnings` before you call one finished: CI runs all three.

Notes, scales and chords

1. A chord that is still missing (lineup-core, easy)

`ChordQuality::ALL` holds thirty qualities, from the triads to `Dominant13` and `Major13` (see [Chords: building, naming and inversions](#)). Read `ChordQuality::intervals`, `symbol`, `name` and `numeral_suffix`, then add one that is absent, such as the minor-major ninth, `MinorMajor9` (0, 3, 7, 11, 14), to `ChordQuality::ALL` too, whose length, part of its type, goes up to 31. The compiler lists every `match` to update. `SPELLINGS`, in `harmony/symbol.rs`, needs a line for the new quality, and `every_quality_is_written_and_read_back` fails until its symbol reads back as the new quality. `no_two_qualities_have_the_same_notes` tells you if its pitch set collides with an existing one.

As pitch classes, why do big chords so often look like other chords on another root? Write out the notes of C13 and find a seventh chord hiding in them.

Practises: exhaustive `match`, arrays whose length is part of the type, letting tests guard your data.

2. Cadd2 and the power chord (lineup-core, medium)

Pop charts sometimes write **Cadd2**: the same notes as Cadd9, with the D played low, next to the C. Add “add2” to the spellings of `Add9` in `SPELLINGS`, and a test that reads it. What is lost by reading it as `Add9`?

C5 is a **power chord**: just C and G, no third, the guitar’s favourite. It is not a `ChordQuality`. Before adding it, list what else would have to know about it: `ChordQuality::family`, which every chord must answer as major, minor or diminished; `is_minor`; the harmony keys; and `find_chord`, which would then name every bare fifth. Is a chord without a third major or minor? Decide, and write the decision into a test.

Practises: following a type through the program before changing it.

3. The line cliché (lineup-core, small)

The **line cliché** keeps one chord while a single voice moves down by semitones: `Am | Am/G# | Am/G | Am/F#`. Read it with `library::chart::parse_chart_in`, play it with `chart::accompany` and `Bass::Held`, and check the bass notes in a test. Is G# in A minor? In which minor? And what is Am/F#, as a chord of its own, in the names of [Chords](#)?

Practises: reading chart text, slash chords, a test on music you can sing.

4. A drop-2 voicing (lineup-core, medium)

`tools::inversion(pitches, times)` and `tools::invert(notes, up)` turn chords over (see [Editing tools as plain functions](#)). Jazz pianists and arrangers use another voicing too, **drop 2**: take a chord in close position and lower its second-highest note by an octave. C E G B, from C4 up, becomes G3 C4 E4 B4.

Add `tools::drop_two(notes)` beside `invert`, with a test like `inverting_chords_up_and_down`, and return `None` when a note would fall below `Note::MIN`. Then put it in the piano roll’s **Tools** menu. Does the chord lane still name the chord the same way, or does the new bass change its name? Why?

Practises: `Vec` manipulation (`remove`, `insert`, `sort`), tests that check your own output.

5. Snap to scale, with a direction (lineup-core and lineup-app, medium)

`tools::nearest_in_key` and `tools::snap_to_scale` move notes outside the key to the nearest note of it, for the piano roll's **Snap to scale**. `nearest_in_key` tries one semitone below, then above, then two, so the lower note wins a tie, and in C major C# goes to C.

Add `nearest_in_key_towards(note, scale, up)`, which breaks a tie in the direction the melody is moving: a C# on its way up to D should become D. Write tests for notes exactly halfway between two notes of the key, both ways.

`snapping_moves_only_notes_outside_the_key` pins the old rule; would the new function change its answers?

Practises: searching with iterators (`find_map`), keeping the logic out of the window.

6. Minor numerals, two ways (the app and lineup-core, easy)

Edit > Settings... > Harmony > Roman numerals in minor keys writes the numerals of a minor key in two ways (see [Minor keys: bIII or III](#)).

1. In A harmonic minor, what does each way write for C augmented (C E G#)? And for E minor? Work it out, then check with the harmony keys.
2. Dorian is a minor mode too, but `Scale::is_minor_key` is false for it, so its numerals always count from the major scale. D Dorian is D E F G A B C. Which of its numerals would change if they counted from its own scale, and why might that confuse more than help?
3. `MinorNumerals::with` keeps the old setting in a guard, `let _restore = Restore(MinorNumerals::current());`, whose `drop` puts it back. Rename `_restore` to `restore` and run `cargo clippy`. Then change it to a bare `_` and run `cargo test -p lineup-core`. What happens each time, and why does `_` behave so differently from `_restore`?

Practises: reading numerals, guards that clean up in `Drop`.

7. Think about it: naming chords

- `find_chord` tries the bass note as the root first. Can you find a set of notes where this gives a surprising name? Is there a better rule?
- When nothing fits, `find_chord` adds the missing fifth and tries again, but only accepts chords of four notes or more. Why would accepting a triad that way be

wrong? Think of C and E alone.

Numerals and what chords do

8. The Neapolitan comes next (lineup-core, medium)

After a chord, the harmony keys outline the chords that often follow it, from the tables in `harmony/progression.rs` (see [What comes next: knowledge as data](#)). The chord sheet already names the **Neapolitan**, $\flat\text{II}$, a major chord on the lowered second degree ($\text{D}\flat$ in C), but `next_chords` has no slot for it, so after a $\text{D}\flat$ chord nothing lights up.

A Neapolitan usually goes to V, or to $\text{I}\flat_4$ and then V. Add a `Slot` with root 1 and `ChordFamily::Major`, with a move to V and a reason, to `MAJOR` and to `MINOR`. Write the tests first: after $\text{D}\flat$ in C major, G; after $\text{D}\flat$ in C minor, G too. Is $\text{D}\flat$ a note of C minor's scale, or is the Neapolitan outside that key as well? A `Suggestion` is only made for a chord whose root is a note of the scale: find the field of `Suggestion` that needs it, and explain why the Neapolitan itself can never be suggested.

Practises: extending a data table, tests as a specification.

9. Functions by ear (the app, easy)

Type these on the chord sheet and listen.

1. `C Fm C`, then `C F C`. The only note that changes is A to $\text{A}\flat$. What does it do to the mood? Find a song you know that does this.
2. In G major, which chord is V7/V ? Which of its notes is not in G major, and where does that note want to go? Type it and check the function the sheet gives it.
3. Write a modulation from C major to F major through a pivot chord. Which chords do the two keys share? Which chord shows that F major has arrived? (Which note does F major have that C major does not?)

Practises: hearing borrowed chords, secondary dominants and pivots.

10. Spelling decides (lineup-core, small)

1. `is_neapolitan` checks the chord's spelling with `key.tonic_name().letter.steps_up_to(symbol.root.letter) == 1`. Explain why C# major in C fails that test while D♭ major passes, though both are played on the same keys of the piano. Find the test that says so.
2. The chord sheet does not name the **augmented sixth** chords (Italian, French and German), which lead to the dominant too. Look up their notes in C minor. Which of them sounds exactly like a dominant seventh on A♭, and what in its spelling tells the two apart? What would a `Role::AugmentedSixth` have to check?

Practises: why music spells notes, letters as well as semitones.

11. A row of borrowed chords (lineup-app, medium)

Songs in major keys often borrow chords from the minor key on the same tonic: iv (F minor in C), ♭VI (A♭), ♭VII (B♭) and ♭III (E♭). This is **modal interchange**, behind many of pop's bittersweet moments, and the chord sheet already names such chords as borrowed. The harmony keys, though, play only the key's own chords.

Add a second, smaller row of keys under the harmony keys with these four chords, built from numerals with `library::numeral::parse` and `RomanChord::in_key`. Give them their own colour, and a tooltip that says which minor-key chord each one is.

Practises: reusing the library's numerals, egui layout.

12. Think about it: tables or rules

`next_chords` keeps its knowledge in tables of root, family and moves. Most of the table could be computed from a few rules instead: roots falling a fifth, steps up to the dominant. Which entries would the rules miss? Is a table that says plainly what it means better than rules that are shorter?

Editing tools

13. Humanize velocities (lineup-core and lineup-app, easy)

Notes all played exactly as hard sound mechanical. The demos already vary their velocities (`library/interpret.rs`), but the piano roll has no tool for your own notes. Add `tools::humanize(notes, amount, seed) -> Vec<NoteEvent>`, which moves each velocity up or down by at most `amount`, kept within 1 to 127, and a “Humanize velocity” item in the piano roll’s **Tools** menu.

Ferrous Lineup has no random number crate, and does not need one:

`ToneRow::random` and the soloist use small generators of their own, and `library/interpret.rs` uses the “SplitMix64” mixer, a few lines of `wrapping_mul` and shifts. Why does the function take a `seed` instead of making one up? How would you test it, and what should the same seed give twice in a row? Where would the piano roll get a different seed each time?

Practises: pure functions with an explicit seed, `clamp`.

14. Retrograde a phrase (lineup-core and lineup-app, medium)

Playing a phrase backwards is a classic way to make a variation. Rows have their retrograde (see [Twelve-tone rows and their rules](#)), but the piano roll has no tool for a phrase. Add `tools::retrograde(notes: &[NoteEvent], span: usize) -> Vec<NoteEvent>`: within `span` steps, each note keeps its pitch and length, and its place is mirrored.

First decide what “mirrored” means for a note of length 2 in a span of 16: does its start mirror, its end, or both? Which choice keeps the phrase inside the span and sounds most like the original played backwards? What happens to `nudge` and `trim`? Add a “Retrograde” item to the **Tools** menu, with a test on a four-note phrase you can sing.

Practises: deciding a musical rule before coding it, `NoteEvent` fields.

15. Note-length tools (lineup-core and lineup-app, medium)

The piano roll’s **Length** sets how long *new* notes are, but no tool changes notes already written. Add two:

- `tools::set_note_length(notes, steps)` makes every note `steps` long, keeping its start and pitch;
- `tools::legato(notes, pattern_length)` makes each note last until the next note starts, so a line joins up.

`NoteEvent` keeps its length in whole steps, plus `trim`, how many ticks before the end of its last step it stops. A note made longer must clear a `trim` that no longer means anything, and `tidy_notes` keeps a pattern's notes in order and inside it. Check what happens when two notes of a channel would then overlap. Why is `set_note_length` not simply `note.length = steps`?

Practises: note lengths and ticks, keeping a pattern's rules.

16. A strum tool, or a channel strum (all three crates, harder)

A guitarist strikes a chord one string after another. The band's guitar does it already: `strum` in `band/guitar.rs` starts each string a few ticks after the one before. Your own notes cannot do it yet. Work out two designs before writing any code.

- **In the notes.** A Strum tool gives each note of a chord a growing `nudge`, from the lowest note up, a few ticks apart. Nothing else in `NoteEvent` changes: the file saves `nudge` already, and Quantize puts the notes back on the grid. A nudge is at most 23 ticks, one step less one.
- **In the instrument.** Each channel gets a "strum" setting in milliseconds. When several notes of a channel start in the same frame, the audio thread starts them that far apart, lowest first. Nothing in the pattern changes, but `Channel`, the `InsertChannel` command, the app's `project_commands` and the file format all need the new field.

Which one lets you strum one chord and not the next? Which one changes fewer files? Real studios have both.

Practises: designing across crates.

17. Think about it: tools

- The selection holds notes by value, which works because a channel never holds two notes in the same place: `same_place` compares pitch, start and `nudge`. Imagine two notes in exactly the same place. What would go wrong with the selection and with `replace_notes`? Would a permanent id for every note fix it, and what would ids cost in the file?

- `tools::transpose` refuses to move a chord if one note would leave the MIDI range. Another design would move the notes that can move and leave the rest. When would that be better? How could the piano roll tell you why nothing happened?

Hearing harmony and finding the key

18. A walking bass, by hand (pencil and paper, easy)

[Hearing harmony](#) works out how C and C7 score over a walking bass, with the passing B $\flat$ weighing 0.15 a step, and says that at 0.35 C would still win, by 0.941 to 0.925. Work those two numbers out yourself: the B $\flat$ then weighs 1.4 and the profile 29.4. Then change `PASSING_BASS` in `harmony/hearing.rs` and write the case as a test. At what weight does C7 win?

Practises: following a scoring rule by hand before trusting it.

19. When to cut a bar (pencil and paper, easy)

When the chord sheet is filled from the notes, a cut costs 0.08 halfway through the bar (`CUT_HALF_BAR`) and 0.12 on any other beat (`CUT_BEAT`) (see [Cutting a bar into chords](#)). A bar of 4/4 plays C, then G7. Each of them, heard alone, scores 1.02 per beat; the whole bar, heard as one stretch, 1.0 per beat.

Does the sheet cut the bar when G7 comes on beat 4, for one beat? And when it comes on beat 3, for two? Look at the `if total > here.0` in the loop: what happens on a tie? How high must the score of each chord alone be for the cut to happen in each case?

Practises: dynamic programming on a small case.

20. The Augurs chord (the app and pencil, easy)

Play E, G $\sharp$ and B with the left hand and E $\flat$, G, B $\flat$ and D $\flat$ with the right: the chord that opens the Augurs of Spring in *The Rite of Spring*. Now play the A $\flat$ harmonic minor scale, A $\flat$ B $\flat$ C $\flat$ D $\flat$ E $\flat$ F $\flat$ G, and check that every note of the chord is in it. Why does `in_one_key`, in `harmony/hearing.rs`, leave harmonic minor out?

Then a question about the cache of `write_heard_chords`: its key holds the notes of the bars around a bar too, not only the bar's own. Why?

Practises: enharmonic spelling, what a cache key must include.

21. Measure the library's ear (lineup-core, medium)

The library has a simpler ear of its own, `library::listen::hear_chords`, for the band's patterns, and `every_style_is_heard_as_its_chart` checks it on every style as written. Write a report of your own, an ignored test: loop over the styles, play each chart with every `Comping` and a few `Bass` lines through `chart::accompany`, hear it, and count how often the chords come back as written. Print the combinations that fail most. Then try to raise the number by changing one constant at the top of `listen.rs`, keeping the existing test passing. Run it with `cargo test --release`: the same test runs many times faster.

Practises: measuring before changing, nested loops over enums.

22. Where the Habanera changes key (the app, easy)

1. Open the Habanera demo (Bizet's, from *Carmen*), turn on 🔍 **Find the key**, and click bars 9, 10 and 11 on the ruler. Where exactly does D minor turn to D major? Is the D major refrain just "the same tune in another key"? Compare the first notes of both.
2. Hold A, C and E on the typing keyboard for a few bars, then add G#, the raised seventh. Does the listener become surer of A minor? Why does one note matter so much?

Practises: hearing a modulation, what makes a key a key.

23. The price of a key change (lineup-core, easy)

In `harmony/key_finding.rs`, set `CHANGE_COST` to 0.8 and run the report:

```
cargo test --release -p lineup-core --test key_finding report -- --ignored --nocapture
```

How many changes are found now, and how many of them are not in the scores? Try 3.0. What does that tell you about the penalty?

Practises: tuning by measurement.

Sets, rows and micromodes

24. A tour of the Row window (the app, easy)

1. Open **File > Demos > Twelve-tone > Triad chorale** and press Space. Watch the section names in the playlist: the same music, then backwards, upside down, and both.
2. Open the piano roll (**F7**) and click **Row....** Pick **Famous and example rows > Schoenberg's Suite**. What does the window say about its groups of three, and which form is it combinatorial with?
3. Click the label **I11** at the top of the matrix, set **Together** to **6: hexachords**, and click **Write I11 into current pattern**: two hexachords as chords. Press **Ctrl+Z** to take it back.
4. Tick the three **Serialism** boxes, set **Octaves** to **Wide**, and write P4 again: Webern-like points of sound.
5. Click **Sets...**, type `octatonic`, and click **Use as the key**. The harmony keys now play chords from the octatonic scale.

25. Distinct forms (lineup-core, small)

A row has 48 forms, but they are not always 48 different rows. Write a test that collects them into a `HashSet<ToneRow>` and counts them. Check that Webern's Symphony row has 24 and Schoenberg's Suite 48. Can any row have fewer than 24? (Could a row be one of its own I forms? Write out what that means for each note, and you will find that at most two notes could fit.)

Practises: `HashSet`, a proof by counting.

26. Derived rows (lineup-core, medium)

Webern's Concerto, Op. 24, uses a row whose four groups of three all belong to the same class, 3-3 [014] (see [Micromodes](#)). Such a row is called **derived**. Add `ToneRow::is_derived(size)`, true when every segment of that size has the same set class, and a `random_derived` that builds one by backtracking, as `random_all_interval` does.

Practises: backtracking, `segments` and set classes.

27. Forte's way (lineup-core, small)

For six classes, Forte's prime form differs from Rahn's, which Ferrous Lineup uses (see [Pitch-class sets](#)). Write `forte_prime()`, which breaks ties from the start of the set instead of from the end, and show it in the Sets window's details when it differs. Why can the trick of "the smallest number" not do this one directly? (Think about which bits a comparison of numbers looks at first.)

Practises: bits as sets, reading a definition carefully.

28. Multiplying by 5 (lineup-core, medium)

Multiplying every pitch class by 5 (mod 12) turns the chromatic scale into the circle of fourths: 0 5 10 3 8 1... This **M5** operation is another way to transform a row or a set, used by some composers after Webern. Add `PitchSet::multiply` and `ToneRow::multiply`, and find which set classes M5 turns into a different class. (The Z-related pairs are worth a look.)

Practises: modular arithmetic, testing over every set class.

29. A series of its own (lineup-core and lineup-app, medium)

With **Durations** ticked, the row writes its own numbers as the durations. Let the user type a second row, used only for the durations, and show both series in the window.

`RowWriting` derives `Copy`: why does it still compile once it holds a second `ToneRow`, and why would it not if the second series were a `Vec<u8>`?

Practises: `Copy` and what it asks of every field.

30. The twelve-tone rules by ear (the app, easy)

1. Choose **twelve-tone row** as the key, pick **Schoenberg's Suite** in **Row...** with E as the key's note, and write P4 (E F G C# F# Eb Ab D B C A Bb) as a melody. Tick **Voice leading**. Then swap two notes, then leave one out. How many notes does the check mark each time, and where does the dashed outline go?
2. Write the first three notes of P4 as a chord, then the rest as a melody. Then write notes 1, 2 and 4 as the chord. What changes?
3. Write P4 in one track and P4 an octave lower in another, in the same rhythm. What does the check say, and why does it say it only once?

4. Write a chord of C, E and G in a row key. Then G, B, D and F. Which rule do they break, and why did Berg break it on purpose?

31. A repeated note (lineup-core, small)

In `walk::take`, in `harmony/voice_leading/serial.rs`, a note that only repeats the note just played is never out of place. Find the lines that make it so. Then write a test where a chord shares a note with the chord before it, and check that nothing is marked.

Practises: reading bit masks, a test for an exception to a rule.

32. A tour of the micromodes (the app, easy)

1. Open the piano roll (**F7**) on the starting song's **Chords** pattern, press **Ctrl+A**, and click **Micromodes...** Read the colours of C major: eight Mayor 2, six triads, one diminished.
2. Click **Menor 3**, then ► **Hear it**. Try **Inverted (I)** and **Move it up**, and watch the triangle turn and flip on the clock.
3. Click **3-3 in Sets** →. The Sets window opens on 3-3. Pick **7-35** and click one of the micromodes under **Micromodes inside** to come back.
4. Open **File > Demos > Micromodes > Triads, two notes kept** and press Space. Listen for the one note that moves in each chord.
5. Click **Webern's trichords** in the Library's **Twelve-tone rows** group. With **3 notes** chosen under the key, the harmony keys play its four groups of three, each labelled 3-3. Now switch the key menu from **twelve-tone row** to **pitch-class set**: the key becomes the row's first group of three, and the harmony keys play its forms.
6. Back on the starting song, while it plays, click **Fourths groove** in the Library: the song goes on in sus chords.

33. The micromodes of a melody (lineup-core, small)

Write `melody_micromodes(notes: &[PitchClass]) -> Vec<Option<&'static Micromode>>`, which names the micromode of every three notes in a row: notes 1 to 3, 2 to 4, 3 to 5 and so on (`windows(3)` does the walking). Why is the answer `None` for some windows? Test it on Webern's Concerto row: its four groups are Menor 3, but what are the windows that straddle two groups?

Practises: `windows`, `Option` in a list.

34. Every colour at once (lineup-core, small)

Loop over `SetClass::all()` and find the smallest sets whose micromode content has no zero: sets that hold all twelve micromodes. How many notes do they need, and which are they? (For the size: the augmented triad and Menor 1 must both fit.)

Practises: searching a table, `micromodes_in`.

35. Schat's tonalities (lineup-core, medium)

Webern's Concerto row splits the twelve notes into four forms of Menor 3. Write a backtracking search that finds every way to split the twelve notes into four forms of one micromode, and count them for each of the twelve. One micromode cannot do it at all: which, and why? (Think about which notes the forms of each micromode can take.)

Practises: backtracking, counting carefully.

36. A walk that remembers more (lineup-core, medium)

The walk of a micromode style that keeps two notes picks the freshest new note first, and only then the form used least recently (see [The walk: choosing with a tuple](#)). Make a walk that is sure to use every form it can reach before any comes back, and test that Menor 3 visits all 24 of its forms in 24 steps. Is that always possible? For which micromodes can a "keep two notes" walk visit every form exactly once? (This is a question about paths in a graph.)

Practises: graphs, backtracking.

37. Hear the family (lineup-app, small)

Add a ► next to each family's name in the Micromodes window that plays its micromodes one after another, from middle C. `MicroAction::Hear` plays notes together; what would you add to play them in turn? Look at how the piano roll's preview of a chord is stopped.

Practises: actions as data, timing in the window.

Voice leading

38. Parallel octaves (the app, easy)

Write I–IV–V–I in C major in four parts, the bass C3 F3 G3 C3 and the soprano E5 F5 D5 E5. Tick **Voice leading**. Then move the soprano's D5 to G5 and read what the check says. Why are octaves between the soprano and the bass avoided, while thirds and sixths between them are welcome?

39. The leading tone and the seventh (the app, easy)

1. Write G7 to C with the seventh, F, in the alto falling to E. Then write G7 to Am, the deceptive cadence. Where must the leading tone go now? And the seventh?
2. The check leaves the leading tone alone in inner parts. Write V7 to I with the tenor going from B down to G. What does the C chord gain, and what would it lack if the tenor rose to C instead?
3. Write C, F, G7, C in four parts with the bass C3 F2 G2 C3, every inner part moving by step or holding. Read the line of moves at each change: are the outer parts ever in parallel or similar motion?

40. A unison to an octave (lineup-core, small)

In `kind_of_move`, a unison going to an octave falls through to the last two arms of the `match`. Work out which kind it gets when both parts rise, then write a test in `voice_leading.rs` that checks it, with the `satb` helper the other tests use.

Practises: reading a `match` arm by arm.

41. A step of three semitones (lineup-core, small)

`Move::size` calls one or two semitones a step. An augmented second, A $\flat$ to B, is three semitones, and its name says “second”. Which size does the code give it, and which would a harmony teacher give it? Change `size` to count letters, as `motion` does, and write a test. Which other moves change size?

Practises: letters and semitones, again.

Importing a recording

42. A sus4 template (lineup-listen, medium)

`chords.rs` knows six qualities. Add `ChordQuality::Sus4` to `QUALITIES` and see what it does on a recording with a sus chord in it: render a demo with one to WAV (the `write_wav` test in `tests/key_changes.rs` does that). More candidates means more ways to be wrong. Does `SEVENTH_COST` need a cousin for rare chords?

Practises: measuring a change on known data.

43. Six-eight (lineup-listen, medium)

`meter::guess` knows 3 and 4 beats to the bar. Add 6/8: six beats in two groups of three, where the chords change on beats 1 and 4. How do you tell 6/8 from 3/4 at twice the tempo, when both put changes every three beats? (Perhaps you cannot, from chords alone: what else could tell them apart?)

Practises: the limits of a method.

44. A recording without a copy (lineup-audio, lineup-core and lineup-listen, harder)

The import saves a copy of the recording with a lead-in of silence (see [Rough edges](#)), only because a sampler note plays its sound from the first sample. The sampler can already start a voice partway (`start_at` , used by `resume` , see [Starting partway](#)). Give `SampleSource::File` a start offset in seconds, negative for silence before the sound, and let the sampler's `note_on` start there. Then delete `write_wav` , and set the offset from the lead-in instead. What must `resume` add to `elapsed` ? And `SampleSource` is saved in project files: which format number does the change need, and what does an older file read as?

Practises: changing a type that is saved in files, the audio thread.

45. A clear key, or the whole recording's (lineup-listen, easy)

`keys::main_key` takes the key of the longest stretch, even when that stretch's key is not clear (`KeySpan::is_clear` , see [Key changes](#)). Try two other rules: the key of the

longest *clear* stretch, and `bpm-key` 's key of the whole recording (`whole` in `analyze`) whenever the longest stretch is not clear. Run the report in `tests/key_changes.rs` for each. Which rule gets the most songs right, and are the songs it gets wrong the same?

Practises: letting a measurement decide.

46. Think about it: listening

- Chords are named per beat, and a beat's chroma is the average of its frames. What happens to a chord that changes on the "and" of a beat, as in a lot of jazz and Latin music?
- Why is the meter guessed from the chords and not from the loudness of the beats, the way a listener feels the downbeat?
- The import runs on its own thread and is let go, not stopped, when another project opens. When would it be worth stopping it, and how would you tell a thread to stop? (See how an export is stopped.)

The library: styles and rhythms as data

Files: `crates/lineup-core/src/library/`, `crates/lineup-core/src/sources.rs`, `crates/lineup-app/src/library_browser.rs` and `crates/lineup-app/src/app/library.rs`

A musician who sits down to play a bossa nova knows four things: the chords, how the guitar plays them, what the bass does, and the drum pattern. None of it is code. It is *knowledge*, and the best way to put knowledge into a program is to write it down as data, in a form a person can read and check, and to keep the code that uses it small.

The library does that: 215 harmony styles, 139 rhythms and 6 fills, each a few lines in a table, plus the twelve-tone and micromode styles of the theory chapters and the songs of [How a demo is written](#). This chapter goes through the notations, then follows a style from the table into a song: which channel plays what, how a whole song changes style, how each part of the song remembers where it came from, and the Library tab that lists it all. How to use the Library is in the manual's [Styles and the Library](#).

Chord charts: chords by the bar

A progression written as chord names, “C G Am F”, only works in C. Written as Roman numerals, “I V vi IV”, it works in every key (see [Reading a numeral back](#)). A **chord chart** writes the numerals bar by bar, the way a lead sheet does, with `|` between bars:

```
I | IV V | vi | ii7 V7
```

The chords in a bar share it equally, so “IV V” is half a bar each. A chord that repeats the one before is held on, not played again: “I | I” is two bars of one chord.

`chart::parse_chart(text, bar)` turns a chart into `ChordSpan s`, steps of a bar of `bar` steps:

```
pub struct ChordSpan {
    pub chord: RomanChord,
    pub start: usize,
    pub length: usize,
    /// Whether the band plays it early (see [`Push`]).
    pub push: Option<Push>,
    /// Whether the band stops playing its style here (see [`Stop`]).
    pub stop: Option<Stop>,
}
```

It returns `None` when a chord cannot be read, a bar is empty, or a bar cannot be shared equally between its chords: a table entry that is wrong fails a test instead of playing nonsense. `push` and `stop` are the marks of [Shots, holds, rests and pushes](#).

Comping: how the chords are played

The same four chords can be a ballad, a reggae tune or a house track. What changes is the rhythm the chords are struck in, which jazz musicians call **comping** (short for accompanying), and what the bass does. Each way of comping is a short table of **hits**: when a note starts, how long it lasts, which notes of the chord it plays, and how hard.

```
struct Hit<T> {
    /// When it starts, in steps from the start of the pattern's period.
    at: usize,
    /// How long it lasts, in steps.
    len: usize,
    what: T,
    velocity: u8,
}
```

For chords, `what` is a `Voices`: `All` the notes, `Upper` (every note but the lowest, the lighter up-stroke of a strum) or `Nth(n)`, one note counted from the bottom, for arpeggios. For the bass it is a `Tone`: the root, the fifth, the octave, the third, the seventh, the next chord's root played early (an **anticipation**) or the note a semitone from it, leading in. One generic struct serves both, with `T` filled in twice.

```
Comping::Pulse => (4, &[hit!(0, 2, All, 96), hit!(2, 2, All, 78)]),
Comping::Quarters => (4, &[hit!(0, 2, All, 100)]),
Comping::Offbeat => (4, &[hit!(2, 1, All, 92)]),
```

The first number is the **period**: the hits repeat every 4 steps, a beat. `Pulse` is steady eighth notes, the second one softer; `Offbeat` is the reggae skank, one short chord on the second eighth of every beat. The habanera's long, short, long, long takes 8 steps:

```

Compiling::Habanera => (
  8,
  &[
    hit!(0, 3, All, 96),
    hit!(3, 1, Upper, 74),
    hit!(4, 2, All, 86),
    hit!(6, 2, All, 84),
  ],
),

```

macro_rules!, and why not a function

`hit!` is a small **declarative macro**:

```

macro_rules! hit {
  ($at:expr, $len:expr, $what:expr, $velocity:expr) => {
    Hit {
      at: $at,
      len: $len,
      what: $what,
      velocity: $velocity,
    }
  };
}

```

A macro is code that writes code. `macro_rules!` matches what it is given against a pattern, here four expressions (`$at:expr` and so on), and pastes them into the template on the right. `hit!(0, 2, All, 96)` becomes `Hit { at: 0, len: 2, what: All, velocity: 96 }` before the compiler checks anything.

Why not a `const fn hit(...) -> Hit`? Look at what the `match` returns: `&'static [Hit<Voices>]`, a borrowed slice that lives for the whole program. Writing `&[ ... ]` of plain values gives exactly that: the compiler sees an expression made only of constants, puts it in the program's read-only data, and lets the borrow live for ever. That is called **constant promotion**. It only happens for plain values: a *function call* inside the brackets, even a `const fn`, is not promoted, and the compiler says the borrowed temporary is dropped at the end of the statement. The macro keeps the table short to write and still made of plain values. The comment above the macro says so, so nobody “simplifies” it into a function.

Styles: a table in a static

A style puts it together: a chart in a mode, a way of comping, a bass line, a usual tempo and the rhythm that goes with it.

```
pub struct Style {
    pub name: &'static str,
    pub genre: Genre,
    /// The mode the chords are written in: the project's scale becomes
    /// this mode when the style is used.
    pub mode: ScaleKind,
    pub time: TimeSignature,
    /// The progression, as a chord chart (see [chart::parse_chart]):
    /// numerals counted from the tonic, bars separated by `|`.
    pub chart: &'static str,
    pub comping: Comping,
    pub bass: Bass,
    /// The tempo the style is usually played at, in BPM.
    pub tempo: f32,
    /// The name of the rhythm in [super::RHYTHMS] that goes with it, or
    /// "" for none.
    pub rhythm: &'static str,
    /// One or two sentences on what to listen for.
    pub about: &'static str,
}
```

Every field is `&'static str`, `Copy` or a small enum, so a `Style` is `Copy` and the table needs no allocation at all. An entry reads like a sentence:

```
style(
    "Four chords",
    Pop,
    Major,
    "I | V | vi | IV",
    Comping::Pulse,
    Bass::Pulse,
    110.0,
    "Pop beat",
    "The four chords behind hundreds of hits: home, the dominant, the sad
relative minor, \
    and the subdominant.",
),
```

`style` is a `const fn` that fills in 4/4, with `#[allow(clippy::too_many_arguments)]` because nine arguments in a table row are clearer than nine field names. The styles in another time signature add `.in_time(THREE_FOUR)`, another `const fn`, which uses **struct update syntax**:

```
const fn in_time(self, time: TimeSignature) -> Style {
    Style { time, ..self }
}
```

`..self` means “every other field as in `self`”. The table itself is a `static`:

```
pub static STYLES: &[Style] = &[
```

A `const` is pasted into every place that uses it, like a macro; a `static` is one value at one address for the whole program. For a big table used in many places, `static` keeps one copy, and a `&'static Style` pointing into it is as cheap to pass around as a number and can never dangle.

`Style::render(tonic)` plays a style in any key: the chart becomes spans, each numeral a chord in the key, the chords are voiced one after another, moving as little as possible, and the comping and the bass play them. That is the band’s own work, in [How the band plays a chord sheet](#).

Rhythms, one line per drum

Drummers and drum machines think of a beat as a grid: one row per drum, one column per step. The library writes exactly that, as text:

```
rhythm(
    "Pop beat",
    Pop,
    110.0,
    &[
        (Kick, "X... .... X.x. ...."),
        (Snare, ".... X... .... X..."),
        (HiHat, "x.x. x.x. x.x. x.x."),
    ],
    "Snare roll",
    "The backbeat: kick on 1 and 3, snare on 2 and 4, hi-hat on every eighth note.",
),
```

Each character is a step. `x` is a loud hit, `x` a normal one, `o` a soft **ghost note** and `.` silence. `h` and `l` play the drum five semitones higher or lower, for the open and muted tones of a conga or two toms. Spaces only make the beats easy to see, and `|` separates bars, for rhythms longer than one. “Snare roll” names the **fill**, a one-bar break that ends a section.

One character becomes one hit:

```
fn hit(character: char) -> Option<Option<Hit>> {
    match character {
        '.' => Some(None),
        'X' => Some(Some((120, 0))),
        'x' => Some(Some((96, 0))),
        'o' => Some(Some((56, 0))),
        'h' => Some(Some((96, 5))),
        'l' => Some(Some((96, -5))),
        _ => None,
    }
}
```

An `Option` inside an `Option` says two different things: the outer `None` is “this is not a character a rhythm may hold”, so a line with a typo is refused; the inner `None` is “a step of silence”, a perfectly good answer. Here `Hit` is a `type` alias, `(u8, i32)`: a velocity and a shift in semitones.

The same two letters write the Turkish usuls of the Balkan rhythms. On the darbuka, `l` is the low stroke, **düm**, and `h` the high one, **tek**, and the comments cite where each pattern comes from:

```
rhythm(
    "Aksak",
    Balkan,
    90.0,
    &[
        (Kick, "X... .... X... ....."),
        (Conga, "l... h.h. l... h...h."),
        (Shaker, "x.o. x.o. x.o. x.o.o."),
    ],
    "",
    "The Turkish aksak, 9/8 counted 2 + 2 + 2 + 3, on the darbuka: düm,
    te-ke, \
        düm, tek-tek, the low düm on the first and third beats, the long last
    beat \
        split two and one.",
)
.in_time(TimeSignature::of_groups(&[2, 2, 2, 3], 8)),
```

When a rhythm goes into a pattern, it is laid out a bar at a time in the pattern’s time signature:

```
for start in (0..length).step_by(bar) {
    let steps = &bars[(start / bar) % bars.len()];
    for (step, hit) in steps.iter().enumerate().take(bar) {
```

`step_by(bar)` gives the start of every bar, and `take(bar)` stops a long bar at the bar line, so a 4/4 beat in a 3/4 pattern loses its last beat in each bar instead of drifting across the bar lines. A few rhythms, the ones marked `follows_the_meter`, are written again for every bar's own meter instead, for music whose meter changes all the time (see [An orchestra for music that changes meter](#)).

A rhythm for any time signature

When you change a song's time signature by hand, it needs a rhythm in the new one.

`Rhythm::for_time` asks, in order, until something answers:

```
RHYTHMS
  .iter()
  .filter(in_time)
  .find(same_kind)
  .or_else(|| {
    USUAL
      .iter()
      .filter_map(|name| Rhythm::named(name))
      .find(in_time)
  })
  .or_else(|| RHYTHMS.iter().find(in_time))
  .or_else(|| Rhythm::grouped_for(time, same_kind))
  .unwrap_or_else(|| Rhythm::basic(time))
```

A rhythm of the same kind in the new time (a tango in 3/4 is a tango waltz); else the usual one (a waltz for 3/4); else any in that time; else one with as many of the same notes to the bar grouped its own way (7/8 plays as 2+2+3/8, a rachenitsa); else a plain beat made for that time signature. Each `or_else` runs only if everything before it found nothing. How the song's notes move into the new meter is in [Putting a song in another time signature](#).

That last plain beat has a problem: a `Rhythm` holds `&'static str` lines, and 5/4 is in no table. `Rhythm::basic` makes the text when it is first asked for, and keeps it for as long as the program runs:

```
pub fn basic(time: TimeSignature) -> &'static Rhythm {
  static MADE: Mutex<Vec<&'static Rhythm>> = Mutex::new(Vec::new());
  let mut made = MADE.lock().unwrap_or_else(PoisonError::into_inner);
  if let Some(beat) = made.iter().find(|beat| beat.time == time) {
    return beat;
  }
  let lines: Vec<(Drum, &'static str)> = basic_lines(time)
    .into_iter()
    .map(|(drum, line)| (drum, &*String::leak(line)))
    .collect();
```

`String::leak`, `Vec::leak` and `Box::leak` give up ownership of their memory on purpose: it is never freed, and in exchange you get a `&'static` reference to it. Leaking every time would grow without end, so the beats made are kept in a `static Mutex<Vec<...>>`, and each time signature is made once: a handful of small strings for the life of the program. `Mutex::new` is a `const fn`, which is what lets a `static` hold one. A poisoned lock (a thread panicked while holding it) still holds good data here, so `unwrap_or_else(PoisonError::into_inner)` takes it anyway.

Where the notes go

The tables know nothing about channels. `library/apply.rs` decides where their notes go: chords to a synth called “Keys” (or the channel you selected), the bass line to a synth called “Bass”, the melody of a song to “Lead”, and each drum to the channel that plays it. Missing channels are added, with a piano for the Keys and a finger bass for the Bass as their General MIDI sounds.

```
pub fn chords_channel(project: &mut Project, selected: Option<usize>) ->
Option<usize> {
    let free = |(index, channel): (usize, &Channel)| {
        plays_notes(channel)
        && channel.name != BASS
        && channel.name != LEAD
        && !project.is_locked(index)
    };
    if let Some(index) = selected
        && project
            .channels()
            .get(index)
            .is_some_and(|channel| free((index, channel)))
    {
        return Some(index);
    }
    let other = project
        .channels()
        .iter()
        .enumerate()
        .position(|(index, channel)| channel.kind == ChannelKind::Synth &&
free((index, channel)));
    find_synth(project, KEYS)
        .or(other)
        .or_else(|| synth_named(project, KEYS))
}
```

`free` is a closure kept in a variable, so it can be used twice. A **locked** channel is never picked for chords (see [Song form, sections, variations and locks](#)). The chain at the end

shows the difference between `or` and `or_else`: `other` is already worked out, so `or` takes it as a value; `synth_named` *adds a channel*, so it must only run if nothing else was found, and goes in a closure with `or_else`.

Drums are found by what they are, not by name: the channel whose kind is `Sampler(SampleSource::Builtin(drum))`. A kick renamed “Bombo” is still the kick. A synth called “Bass” sent to an external synth over MIDI is still the bass, since `plays_notes` accepts MIDI Out channels too (see [MIDI](#)).

Playing a song in another style

Trying a style in a pattern of its own is easy. Changing the style of a whole song is harder: “let’s do this one as a reggae” means the same chords, played another way by everyone. But a pattern does not store its chords; it stores notes. So the library first **hears** them, with the second ear of [Hearing harmony](#), and then plays them again.

A rhythm says how it likes its chords played through the style whose drums it is:

```
pub fn feel(rhythm: &Rhythm) -> (Comping, Bass) {
    let same_time = |style: &&Style| style.time == rhythm.time;
    STYLES
        .iter()
        .find(|style| style.rhythm == rhythm.name)
        .or_else(|| {
            STYLES
                .iter()
                .filter(same_time)
                .find(|style| style.genre == rhythm.genre)
        })
        .or_else(|| STYLES.iter().find(same_time))
        .map_or((Comping::Pulse, Bass::Pulse), |style| {
            (style.comping, style.bass)
        })
}
```

The “Salsa” rhythm is the drums of the “Salsa montuno” style, so it plays its chords as a montuno over a tumbao bass. A rhythm no style uses borrows a style of its genre in its time signature, so a rhythm in 3/4 never gets a way of playing meant for 4/4. Why `&&Style`? `STYLES.iter()` yields `&Style`, and `filter` hands its closure a reference to each item: a reference to a reference.

`recomp` plays one pattern again: it hears the chords of its chords channel over its bass, voices them, and lets the new comping and bass play each stretch. Stretches where no chord was heard keep their old notes, so an intro without chords stays as it

was. `play_song` does it for every pattern, and leaves three things alone on purpose: the melody, the patterns of the band (which follow the chord sheet, see [How the band plays a chord sheet](#)), and endings, a pattern whose notes all start on its first step, like a final held chord with a crash. Where the old drums had a fill in their last bar, the new rhythm's fill goes there. A fill is a last bar that plays like no bar before it:

```
// A fill is a last bar that plays like no bar before it.
let fill = length >= 2 * bar && {
    let last = drums_in(project, pattern, length - bar, bar);
    (0..length - bar)
        .step_by(bar)
        .all(|start| drums_in(project, pattern, start, bar) != last)
};
```

A block `{ ... }` is an expression in Rust, whose value is its last line, so it can sit after `&&`; and since `&&` stops as soon as its left side is `false`, the block only runs for patterns of two bars or more. A rhythm in another time signature first re-meters the whole song, bar for bar, so a 4/4 song played as a waltz becomes a song in 3/4.

Which parts to take

The window a style opens in the app asks which parts to take, in three groups: under **Accompaniment, Drums, Chord and bass parts** and **Voices / instruments**; under **Harmony, Chord progression**; under **Timing, Time signature, Tempo** and **Swing**. They are one small struct:

```
pub struct StyleParts {
    pub rhythm: bool,
    pub instruments: bool,
    pub time_signature: bool,
    pub progression: bool,
    /// Take the preset's usual tempo where the meter stays the same. Off
    /// by default: the song keeps its tempo, so its own notes last as
    /// long as before.
    pub tempo: bool,
    pub swing: bool,
    /// Copy the preset's chord, bass and other accompaniment parts (a
    /// rhythm's are those of its matching style). Off, the song's own
    /// stay, as its drums do with `rhythm` off.
    pub matching_style: bool,
}
```

`rhythm` is the **Drums** box and `matching_style` the **Chord and bass parts** box. Every box stands on its own: what is left unticked, the song keeps. In a pattern, each box is one step of `write_style_parts`, and each step touches only its own notes:

```

if parts.time_signature {
  meter(project, index, style.time);
}
if parts.matching_style {
  recomp(project, index, style.comping, style.bass);
}
if parts.progression {
  progression(project, index, style);
}
// ...
if parts.rhythm
  && let Some(rhythm) = Rhythm::named(style.rhythm)
{
  write_rhythm(project, index, rhythm);
}
if parts.instruments {
  style_instruments(project, style, selected);
}

```

`recomp` plays the chord and bass channels again and leaves the drums; `write_rhythm` writes the drum lanes and leaves the rest. So a song can take one style's drums and another's chord and bass parts, in either order. A band keeps the same split in a field of its own, `drums` (see [How the band plays a chord sheet](#)). The `if parts.rhythm && let Some(rhythm) = ...` line is a *let chain*: the block runs only when the box is ticked *and* the style's rhythm is found, with `rhythm` bound inside it.

Voices / instruments gives the style's sounds to the channels that play its parts: Keys and Bass, or the channel picked in the rack. A song imported from a MIDI file has neither; its channels carry the names the file gave them, such as "PIANO MELODIA", "BANDONEON" or "CONTRABAJO". Then `own_players_instruments` gives the style's sounds to the song's own players, each by the family of the instrument it had, and leaves the tune alone:

```

let program = match channel.program? {
  general_midi::CONTRABASS => band.bass_program(),
  // Violins to tremolo strings, and the string ensembles.
  40..=44 | 48..=51 => extras.strings.as_ref()?.program,
  program => match general_midi::FAMILIES[general_midi::family(program)]
{
  "Piano" => piano,
  "Organ" => extras.organ.as_ref().map_or(piano, |organ|
organ.program),
  "Guitar" => extras
    .guitar
    .as_ref()
    .map_or(piano, |guitar| guitar.program),
  "Bass" => band.bass_program(),
  _ => return None,
},
};

```

The code sits in a closure given to `filter_map`, which returns an `Option`, so every `?` and the `return None` mean “this channel keeps its sound”: a channel with no General MIDI program, strings when the style has none, a choir or a bandoneon, which no style has. A double bass is in the General MIDI strings family, so it is matched first, by its number, and takes the bass. The tune is any channel whose name says so (Lead, Melody, Melodía, Voice, Voz, Canto), or else the one the lead sheet would show.

Two small methods correct the choice for the kind of preset, each taking `self` by value and giving back a changed copy: `for_rhythm` turns off the progression, which a rhythm does not have; and `following` keeps the song’s time signatures and tempo for a style that plays every bar in its own meter. They chain:

```
parts.for_rhythm().following(rhythm.follows_the_meter) .
```

The choice is kept between runs, in egui’s memory, under the name `apply-options`. Until you change it, it is everything but the chord progression, so the song keeps its own chords:

```

const FIRST_OPTIONS: StyleParts = StyleParts {
  rhythm: true,
  instruments: true,
  time_signature: true,
  progression: false,
  tempo: true,
  swing: true,
  matching_style: true,
};

```

`#[serde(default)]` on the struct makes a field added later read as its default from an older save. A `const` of a struct is written out in full where it is used, so

`FIRST_OPTIONS` costs nothing at run time.

The same style, the same song

Try Bluegrass, then Celtic, then Bluegrass again: the third should play exactly as the first. A style must not be applied on top of the last one, with the last one's meter and tempo, or every click would drift a little further from the song. So the app keeps, next to each version of the song the Library made, the song as it came, in its own meter:

```
pub(super) struct SongSource {
    original: Project,
    library: Tried,
    /// Latest preset components, applied without changing the source
    meter.
    reference: Project,
}
```

Each click starts again from `reference`, and asks whether the style's meter differs from *that* song's, not from the one the last style put it in:

```
let changes_meter = !lineup_core::remeter::is_in(&source.reference, time);
```

The tempo follows the same idea. **Tempo** puts the style's tempo where most of the song plays, with `take_tempo` (see [The tempo strip](#)): a song that starts with a slow bar and then plays at 121 BPM, given a style made for 180, plays at 180 where it played at 121, and its slow bar stays as much slower as before. Giving the style's tempo to the song's first bar instead would make a MIDI file with a slow lead-in race.

Where each part came from

Taking only some parts means a song can have its drums from one style, its chord and bass parts from another, its tempo from a third and its chords from the demo it started as. The Library shows it, a line per part: Tempo "Tango marcato (style)", Chords "El Choclo (demo)", and **edited, was ...** once you change that part yourself.

`sources.rs` keeps that record.

The nine parts are an enum, in the order the Library lists them: the drums, the chord and bass parts, the melody, the instruments, the time signature, the key, the chords, the tempo and the swing. A set of them is nine bits of a `u16`, one bit a part:

```
pub struct PartSet(u16);
```

```
pub fn contains(self, part: SongPart) -> bool {  
    self.0 & part.bit() != 0  
}
```

Every change that brings parts in goes through one function, which compares the song before and after:

```
pub fn bring_in<R>(  
    &mut self,  
    source: &Source,  
    taken: PartSet,  
    edited: PartSet,  
    change: impl FnOnce(&mut Project) -> R,  
) -> R {  
    let before = SongPart::ALL.map(|part| self.part_fingerprint(part));  
    let result = change(self);  
    for (part, before) in SongPart::ALL.into_iter().zip(before) {  
        let now = self.part_fingerprint(part);  
        // A key that changed with what was taken came with it: a  
        // minor progression puts the song in minor, a row style puts  
        // it in its row.  
        let came = taken.contains(part)  
            || (part == SongPart::Key && now != before && taken !=  
PartSet::NONE);  
        if came && !edited.contains(part) {  
            self.sources.set(  
                part,  
                Some(PartSource {  
                    source: source.clone(),  
                    print: Some(now),  
                }  
            );  
        } else if now != before && !edited.contains(part) {  
            self.follow_part(part, before, now);  
        }  
    }  
    result  
}
```

The parts taken now come from `source`, even if they did not change: a 4/4 style applied to a song already in 4/4 still gives it its time signature. A part that changed only along with them, such as chords moved into a new meter, *follows along*: it still comes from where it came from, and is as it came if it was before. The key is the one part no box names: it comes with whatever brought it, so a minor progression's key comes from its style.

The `print` is a **fingerprint**: a `u64` hash of the part as it came, made with `std::hash::DefaultHasher` from its notes, its tempo, its chords. Whether a part is edited is not stored anywhere; it is asked each time, by comparing the fingerprint with the part as it is now:

```
edited: came.print != Some(self.part_fingerprint(part)),
```

So editing the tempo makes it “edited, was ...”, and setting it back, or undoing, makes it as it came again, with no code to keep a flag up to date. The project file saves only the source and an `edited` flag, never the hash: `DefaultHasher` may change its algorithm in another Rust release, and a stored hash would then call every part of every old file edited. On loading, the print is worked out again from the part as it is.

Three of the parts are notes of the same patterns: the drums, the melody and the chord and bass parts, which are everything else. They are told apart by channel, with one helper that hashes the notes of the channels a closure picks:

```
SongPart::Melody => self.hash_notes(&mut hasher, is_lead),
```

and the chord and bass parts pass `|channel| !is_drum(channel) && !is_lead(channel)`. With a band, the drums and the chord and bass parts are what the band plays, so their fingerprints hash the names of the styles they come from instead.

Older versions knew only six of the parts, and would refuse a file naming one they do not know. So the drums, the melody and the key are saved in a field of their own, `more_sources`, which those versions skip. It is written whenever any part came from somewhere, even empty, so a file without it is from before; for such a file, `sources_of_an_older_file` works out where those three came from: the drums with the patterns, a demo’s melody with the demo, and the key with the chords.

One more detail. The record of sources is not the music, and two songs that play the same should be equal: the app compares projects to know whether there are unsaved changes. So its `PartialEq` says every two records are equal:

```
impl PartialEq for Sources {  
    fn eq(&self, _: &Self) -> bool {  
        true  
    }  
}
```

A hand-written `impl` is how a type takes part in a derived `PartialEq` (the `Project`’s) without changing the answer.

The Library tab

The Library lists harmony styles, rhythms and your own styles in one catalogue, sorted by the names as shown in the language chosen:

```
pub enum Preset<'a> {  
    Style(&'static Style),  
    Rhythm(&'static Rhythm),  
    User(usize, &'a UserStyle),  
}
```

Two kinds of entry borrow from tables that live for ever; your own styles live in the app, so `Preset` has a lifetime, `'a`, which says it cannot outlive the list it borrows from. Every question the list asks (name, time, tempo, whether it brings a progression) is a method with one `match`, so the rest of the browser never asks which kind it is. A rhythm that is the very drums of a style of the same name, tempo and feel is listed once, as the style.

The tick boxes under the search keep only the presets like the song, in its time signature or near its tempo, or those with what you want: a chord progression, drums that swing, an instrument picked from a menu. What they look at in a preset is a small `Copy` struct,

```
struct Traits {  
    time: TimeSignature,  
    /// The speed it was made for, in beats a minute.  
    tempo: f32,  
    /// Whether it plays every bar in its own meter, so it fits any time  
    /// signature.  
    any_meter: bool,  
    /// Whether it brings a chord progression.  
    chords: bool,  
    /// Whether its drums swing.  
    swing: bool,  
}
```

and the decision is one method:


```

fn keeps(self, traits: Traits, instruments: impl FnOnce() ->
Vec<Instrument>) -> bool {
    let filters = self.filters;
    // 7/8 as 2+2+3 is still 7/8.
    let same_time = traits.any_meter || traits.time.ungrouped() ==
self.time.ungrouped();
    let ratio = traits.tempo / self.tempo;
    let near = ratio.max(1.0 / ratio) <= NEAR_SPEED;
    (!filters.time || same_time)
    && (!filters.speed || near)
    && (!filters.chords || traits.chords)
    && (!filters.swing || traits.swing)
    && filters
        .instrument
        .is_none_or(|instrument| instruments().contains(&instrument))
}

```

Read the lines as rules joined by “and”: `!filters.time || same_time` is “either the box is not ticked, or the time is the same”. `ratio.max(1.0 / ratio)` turns “half as fast” (0.5) and “twice as fast” (2.0) into the same number, 2, so faster and slower are judged alike; with `NEAR_SPEED` at 1.15, a song at 130 BPM keeps presets made for 113 to 149.5. Comparing ratios rather than a difference in BPM treats slow and fast songs alike: ± 15 BPM would be a lot at 60 and little at 180. `keeps` takes `self` by value: the struct is `Copy` and a few bytes, so that costs no more than a reference.

The instruments are the one question that costs something: a preset’s are those of the band that would play it, worked out from its style. So they come as a closure, `instruments`, called only at the end, when an instrument is chosen.

`Option::is_none_or` is `true` for `None`, no instrument chosen, and otherwise asks the closure about the one inside. `&&` stops at the first `false`, so a preset already left out by an earlier box never works its instruments out at all.

The list is grouped by genre, one `egui::CollapsingHeader` a genre with its count and your own styles first under **User styles**, unless **Settings > Style library** says **Alphabetical list**. A search opens every group that has a match. When the song changes style, the group of the new one opens for half a second, long enough for `egui` to scroll its lit row into sight, and then leaves the groups as you had them.

Trying, swapping and undoing

What a click does depends on the song. With a band, as in a song from **File > New** or a demo, the style becomes the band’s way of playing the chord sheet (see [How the band plays a chord sheet](#)). While the song plays without a band, the click changes the

whole song, so the music goes on. Otherwise it writes the style into a **trial pattern** of its own and plays it there.

Clicking another style should *replace* the trial, not pile up patterns, so you can flick through the library. But once you start editing that pattern, it is yours. The app tells the two apart by keeping a copy of what it wrote:

```
pub(super) struct Audition {
    pattern: usize,
    /// The pattern as the library left it. If it still looks like this,
    /// the next click may rewrite it.
    written: Pattern,
    style: Option<&'static Style>,
    rhythm: Option<&'static Rhythm>,
    row_style: Option<&'static RowStyle>,
    micromode_style: Option<&'static MicromodeStyle>,
}
```

```
fn unchanged_audition(&self) -> Option<&Audition> {
    let pattern = self.project.pattern(self.pattern)?;
    self.auditions
        .iter()
        .rev()
        .find(|audition| audition.pattern == self.pattern &&
&audition.written == pattern)
}
```

`==` on two `Pattern`s compares every note, thanks to the derived `PartialEq`: instead of tracking every edit, keep a copy and compare, the same idea as unsaved changes. The app keeps the last 64 trials, not only the newest, and searches from the newest with `rev()`. That is for undo: after `Ctrl+Z` brings the previous style back, the pattern matches an older copy, and the next click still swaps it in place.

Harmony styles and rhythms take the parts you ticked (see [Which parts to take](#)). Row styles and micromode styles have no such choice: they are tried the same way, and differ only in the notes they write. So their shared steps are one method, and the different step is a closure passed in:

```
fn write_pattern_style(
    &mut self,
    name: &'static str,
    tempo: f32,
    rhythm: Option<&'static Rhythm>,
    write: impl FnOnce(&mut Project, usize),
) -> Option<(usize, Pattern)> {
```

`impl FnOnce(&mut Project, usize)` accepts any closure that can be called once with those two arguments. The compiler makes a copy of the method for each closure passed to it, so the call costs no more than code written out. `FnOnce` asks the least of the three closure traits (`Fn`, `FnMut`, `FnOnce`): every closure is at least `FnOnce`, so asking for it accepts them all.

Every write goes through `edit`, so a trial is one step to undo, the audio thread hears of it through the usual diff, and saving just works. Inside it, `bring_in` records where the parts came from.

Testing it

The tables are data, and data can be wrong as easily as code, without crashing anything. So the tests read the whole library:

`every_style_pares_and_fits_in_a_pattern`,
`names_are_unique_and_rhythms_exist`, `every_genre_has_styles`,
`styles_play_in_any_key`, and for the rhythms
`every_line_is_whole_bars_of_known_characters`,
`names_are_unique_and_fills_exist` and `congas_play_high_and_low`.
`a_new_time_signature_keeps_the_kind_of_rhythm`,
`an_odd_meter_gets_a_rhythm_grouped_its_way` and
`a_time_signature_with_no_rhythm_gets_a_basic_beat` follow `for_time` down its chain.

`apply.rs` checks where notes go (`styles_go_to_keys_and_bass`,
`chords_go_around_a_locked_channel`), that `recomping_keeps_the_chords`, that a whole song takes a new rhythm or a style, that a waltz puts the song in 3/4, and, the widest, that `every_demo_takes_every_kind_of_library_click`. `sources.rs` checks a part that comes, is edited, and is undone:

```
#[test]
fn a_part_brought_in_is_edited_once_changed_and_not_after_undoing_it() {
    let mut project = Project::demo();
    project.bring_in_all(&demo());
    assert_eq!(from(&project, SongPart::Tempo), Some(("El Choclo",
false)));
    let tempo = project.tempo;
    project.scale_tempo(Tempo::new(tempo.bpm() + 10.0));
    assert_eq!(from(&project, SongPart::Tempo), Some(("El Choclo",
true)));
    assert_eq!(from(&project, SongPart::Swing), Some(("El Choclo",
false)));
    project.scale_tempo(tempo);
    assert_eq!(from(&project, SongPart::Tempo), Some(("El Choclo",
false)));
}
```

For the boxes,

`a_styles_sounds_go_to_an_imported_songs_players_by_family_but_the_tune` builds a song like an imported tango, with a “PIANO MELODIA”, a bandoneon, a guitar, a double bass, strings and a choir, and checks which channels take the sounds of Tango and of Bossa nova and which keep theirs;

`a_songs_keys_and_bass_still_take_a_styles_sounds_alone` checks a song with Keys and Bass is changed only there. In `band.rs`,

`drums_and_chord_parts_come_on_their_own_from_builtin_and_saved_styles` gives a Rock ride band the drums of Bossa nova first, from the built-in style and from a saved copy of it, and checks the piano and the bass stay as they were, the file reads back the same, and once the chord and bass parts follow, the band is plain Bossa nova again, in the format it needed before; `a_styles_chord_parts_can_come_without_its_drums` takes Bossa nova’s chord and bass parts over Rock ride’s drums, which needs format 29. `a_style_tempo_goes_where_most_of_the_song_is` checks `take_tempo` on a song with a slow first bar.

In the app, `the_library_tick_boxes_keep_only_presets_like_the_song` counts the presets listed as the boxes are ticked;

`the_same_style_brings_the_same_tempo_whatever_was_tried_before` clicks Bluegrass, Celtic, Bluegrass and Celtic on a song in 2/4 and in 4/4, and checks each style brings the same tempo every time; and

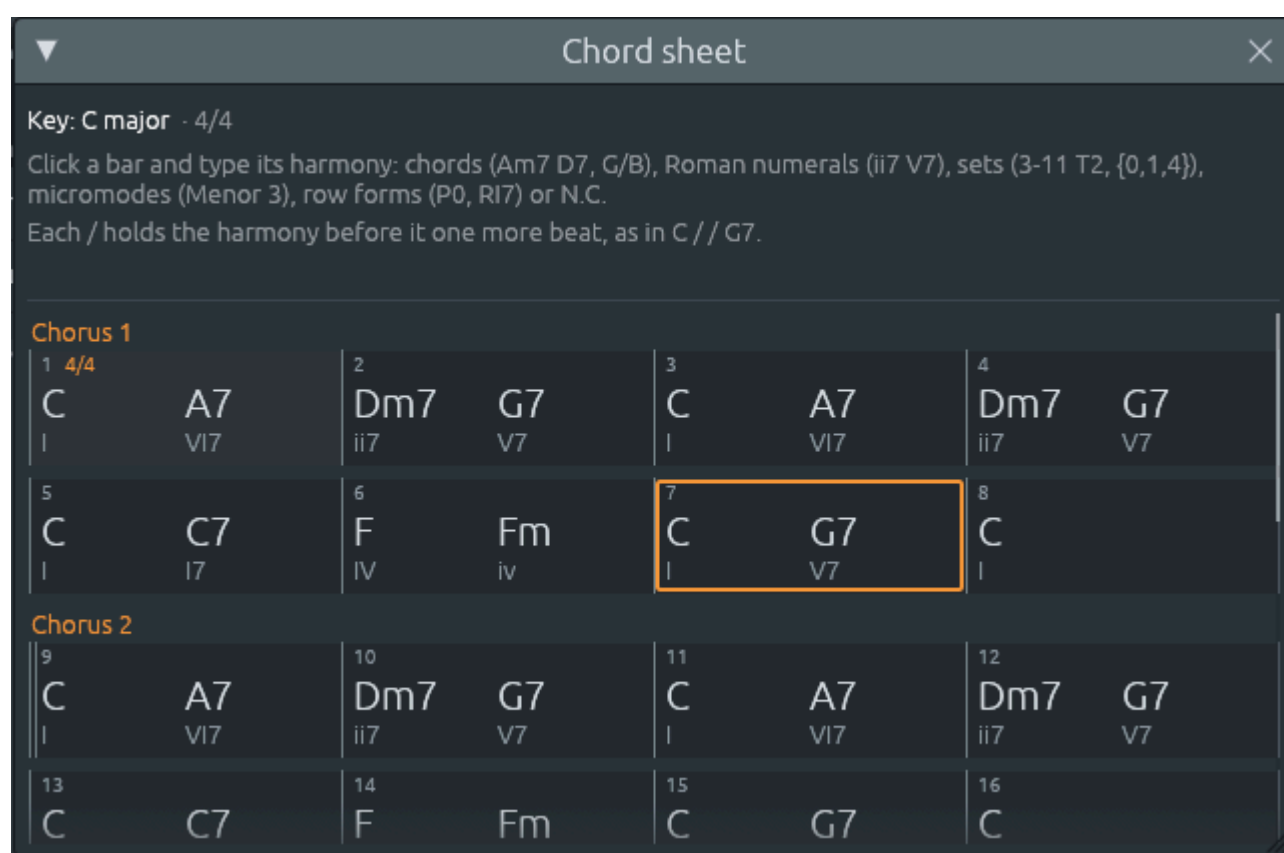
`trying_row_and_micromode_styles_rewrites_one_pattern_while_it_is_unchanged` checks that trials land in one pattern and each is one step to undo.

Exercises on the library are in [Exercises: the band](#).

The chord sheet: model, layout and text

Files: [crates/lineup-core/src/chord_sheet.rs](#), [crates/lineup-core/src/chord_sheet/text.rs](#), [crates/lineup-core/src/bars.rs](#), [crates/lineup-app/src/chord_sheet.rs](#) and [crates/lineup-app/src/chord_sheet/layout.rs](#)

A backing band needs to know the song, and musicians write a song down as a grid of bars with chord names in them: “C”, Tab, “Am7 D7”, Tab. That grid is the **chord sheet**. It opens with **F5**, from the toolbar or from **View > Chord sheet**, as a window over the playlist, or in the playlist’s place in the main window. The band of the next chapters plays from it, and the harmony views of Part III name what is on it.



This chapter follows the sheet from what you type in a bar to how it is stored, where its chords come from, how it follows the song when bars are moved, inserted or deleted, how the window lays the bars out, and how a sheet becomes plain text and back. How to use it is in the manual’s [The chord sheet](#).

What a bar holds

Click a bar and type. A bar holds any of these, separated by spaces:

Typed	What it is	Example
A chord symbol	any chord Chords reads	Am7 , G/B , F#m7b5 , C6/9
A polychord	two chords at once, the upper chord first, joined by	Eb7 Fb , C F#
A Roman numeral	a chord of the key, in a key of seven notes	i i7 v7 in C major is written Dm7 G7
A pitch-class set	by its Forte name, moved with Tn or TnI	3-11 T2 is D F A
	or by its notes, in braces	{0,1,4} , {D F A} , {t e 0}
A micromode	one of Kröpfel's twelve, moved the same way	Menor 3 , Menor 3 T4I
A row form	a form of the key's twelve-tone row	P0 , I5 , R3 , RI7
No chord	nothing plays the harmony	N.C. or NC
A hold	the harmony before goes on one more beat	/ or .
Nothing	the bar goes on with the harmony before	
%	the bar before, again	`

A slash after a chord means a bass note, so the two layers of a polychord are joined by a pipe instead: Eb7 | Fb is Eb7 over Fb major, the chord that opens the *Augurs of Spring*. Chords can be typed in solfège too, “Lam7”, whichever way the app names notes (see [Typing a chord either way](#)), and a harmony can carry the marks of a shot or a push, “C.” or “^C” (see [Shots, holds, rests and pushes](#)).

What you type is written down in one way. A numeral becomes the chord it is in the key, so “V7” typed in Bb major is written F7; a chord is spelled the way the harmony keys spell it, “CM7” becomes Cmaj7; a set typed by its notes becomes its Forte name, {D F A} is 3-11 T2. In a minor key, a numeral you type is read the way the sheet writes numerals, so “VII” in A minor is G, as the setting in [Minor keys: bIII or III](#) writes it. The library's charts are read the other way, from the major scale, because that is how they are written; `read_bar` and `read_typed_bar` differ only in that.

Where the harmonies fall

Lead sheets have a convention for several chords in one bar, and the sheet follows it:

Typed in 4/4	Beats
Am7	Am7 on 1, for the whole bar
Am7 D7	Am7 on 1, D7 on 3: two beats each
C F G	C on 1, F on 3, G on 4: two, one and one
C Am Dm G7	one on each beat
C / / G7	C on 1, G7 on 4
/ G7	the chord before goes on for a beat, G7 on 2

In 3/4, two chords take two beats and one: “C G7” puts G7 on beat 3. The rule behind the even spreading is one line: the `k` th of `n` harmonies in a bar of `beats` beats goes on beat `(k × beats) / n`, rounded up.

```
let beat = if holds {
  place
} else {
  (harmony_count * beats.len()).div_ceil(harmonies)
};
```

`div_ceil` is integer division rounded up, so for three harmonies in four beats the beats are 0, $\lceil 4/3 \rceil = 2$ and $\lceil 8/3 \rceil = 3$. When the bar has any `/` in it, every word is one beat instead (`place` is the word’s position), so you can put a chord on any beat you like. A bar with more harmonies than beats is refused: “this bar has 3 beats, and 4 harmonies are written in it.”

Names of several words

“Menor 3 T2” is one harmony written in three words, and “C Am” is two harmonies in two words, so a bar cannot be split on spaces and each word read alone. Instead, at each word the reader tries the longest run of words first, up to four, and takes the first that reads as a harmony:

```
let holds_at = words[at..]
  .iter()
  .position(|word| matches!(*word, "/" | "."))
  .map_or(words.len(), |offset| at + offset);
let longest = (at + 1..=holds_at.min(at + MOST_WORDS))
  .rev()
  .find_map(|end| {
    read_marked(&words[at..end].join(" "), key, numerals)
      .ok()
      .map(|marked| (marked, end))
  });
```

`(a..=b).rev()` counts down from `b` to `a`, and `find_map` stops at the first `Some`. So “Menor 3 T2 C” reads “Menor 3 T2 C” (not a harmony), then “Menor 3 T2” (a micromode), and goes on from “C”. A run never goes past a `/` standing alone (`holds_at` is where the next one is), because that slash is a beat: “A E / B” is A, E held two beats, then B, not A and the slash chord E/B.

If a word is no harmony at all, the bar stays open with the cursor at the end, and the reason is written above the sheet in red:

Bar 3: “Hm7” is not a chord (Am7, G/B), two chords at once (Eb7|Fb), a set (3-11 T2, {0,4,7}), a micromode (Menor 3), a row form (P0, RI7) or N.C.

Nothing is written into the song until the whole bar reads.

Writing a bar back

Clicking a bar fills its text field with what it holds, so `write_bar(markers, bar)` must give text that `read_bar` reads back as the same markers. It writes the names alone when they fall where the spreading rule would put them, “Am7 D7”, and otherwise a word for every beat, with `/` for the beats that go on, “C / / G7”. A `/` at the end says nothing and is dropped, except when it is the last one left:

```
while words.last().is_some_and(|word| word == "/" )
    && words[..words.len() - 1].iter().any(|word| word == "/" )
{
    words.pop();
}
```

“C G7 C /” puts three chords on the first three beats. Without its slash, “C G7 C” would be three chords spread over the bar, on beats 1, 3 and 4.

How the sheet is stored

The sheet is not a grid of bars in memory. It is a list of **chord markers**, each saying “from this step on, the harmony is this”, like the time, tempo and key markers:


```
pub struct ChordMarker {
    /// The step it starts on, counted from the start of the song.
    pub start: usize,
    pub harmony: HarmonyName,
    /// Whether the band plays it early.
    #[serde(default, skip_serializing_if = "Option::is_none")]
    pub push: Option<Push>,
    /// Whether the band stops playing its style here.
    #[serde(default, skip_serializing_if = "Option::is_none")]
    pub stop: Option<Stop>,
}
```

The harmony is one of the six kinds a bar can hold:

```
pub enum HarmonyName {
    /// A chord symbol: "Am7", "G/B".
    Chord(ChordSymbol),
    /// Two chords at once, one over the other, the upper one first:
    /// "Eb7|Fb" is Eb7 over Fb major, the chord that opens the Augurs of
    /// Spring. Lead sheets draw the two chords as a fraction, with a
    /// line between them; a slash would mean a bass note.
    Polychord(ChordSymbol, ChordSymbol),
    /// No chord, written "N.C.": nothing plays the harmony here.
    NoChord,
    /// A pitch-class set on these very notes, named by its Forte number
    /// and how it is moved from the prime form: "3-11 T2" is D F A.
    Set(PitchSet),
    /// One of Kröpfl's micromodes on these three notes: "Menor 3 T2".
    Micromode(PitchSet),
    /// A form of the twelve-tone row of the key here: "RI7".
    Row(RowLabel),
}
```

A harmony lasts until the next marker, the way a chord on a lead sheet lasts until the next chord is written. That is why a bar with nothing in it needs nothing stored, and why “the harmony at step 200” is a search for the last marker at or before 200:

```
pub fn chord_at(&self, step: usize) -> Option<ChordMarker> {
    let markers = &self.playlist.chord_markers;
    let after = markers.partition_point(|marker| marker.start <= step);
    after.checked_sub(1).map(|index| markers[index])
}
```

`partition_point` does a binary search on the sorted list and returns how many markers start at or before `step`; the one before that index is the answer.

`checked_sub(1)` is `None` when there is none, before the first chord.

Every change to the sheet goes through one method, which writes the harmonies of one stretch of the song, usually one bar:

```
pub fn set_chords(&mut self, start: usize, end: usize, markers: &
[ChordMarker]) -> bool {
```

The markers already between `start` and `end` go, and `markers` take their place. Writing a whole bar at once, rather than adding or moving single markers, matches what you do on the sheet: you type a bar, and what you typed is what the bar holds. Then the list is tidied:

```
pub(crate) fn tidy(markers: &mut Vec<ChordMarker>, end: usize) {
    markers.retain(|marker| marker.start < end);
    markers.reverse();
    markers.sort_by_key(|marker| marker.start);
    markers.dedup_by_key(|marker| marker.start);
}
```

Two markers on one step: the one written last should win. `dedup_by_key` keeps the *first* of each run of equal keys, so the list is reversed before sorting. `sort_by_key` is a **stable** sort, which keeps equal elements in the order they were in, so after the reverse the newest of each step comes first and survives. Undo needs nothing new: the app's `edit` keeps a copy of the project from before each change (see [Undo and redo](#)).

Keeping markers on the timeline, and not a list of bars, lets the sheet share the code that moves the other markers: when a tool shifts everything after some step, the chord markers are shifted in the same loop as the time, tempo and key markers. Which bar a marker falls in is worked out when it is drawn.

In the file

The harmony is saved as the text you would type, not as numbers:

```
chord_markers: [
    (start: 0, harmony: "C"),
    (start: 8, harmony: "A7"),
    (start: 16, harmony: "Dm7"),
    (start: 64, harmony: "Menor 3 T2I"),
],
```

That takes two attributes on the enum:

```
#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash, Serialize, Deserialize)]
#[serde(into = "String", try_from = "String")]
```

`into = "String"` tells `serde` to save a `HarmonyName` by turning it into a `String` first, with `impl From<HarmonyName> for String`, and `try_from = "String"` to load one by

reading a `String` and turning it back with `impl TryFrom<String> for HarmonyName`. `try_from` rather than `from`, because reading can fail: a file with a harmony that cannot be read is refused with the reason, as any broken file is. Both run in letters, whichever way the app names notes, so a file is the same for everyone (see [Always in letters and English](#)):

```
impl From<HarmonyName> for String {
    fn from(harmony: HarmonyName) -> Self {
        NoteNaming::Letters.with(|| harmony.to_string())
    }
}
```

A project with chords on its sheet is saved as format 12 or later, one with a polychord as format 25 (see [The version number](#)).

Where the chords come from

A song opens with its chords already on the sheet, so the sheet is never blank for a song that plays chords.

A **demo built from a chart** writes its own chart, part by part, each part in its own key. El Choclo's third strain, in A major, reads A E7 | E7 A | F#7 Bm | B7 E7: the chart's VI7 ii | II7 V7 named in A major.

```
fn write_chart(project: &mut Project, at: usize, spans: &[ChordSpan], key: Scale) {
    let Some(end) = spans.last().map(|span| at + span.start + span.length)
    else {
        return;
    };
    let markers: Vec<ChordMarker> = spans
        .iter()
        .map(|span| {
            ChordMarker::new(
                at + span.start,
                HarmonyName::Chord(span.chord.symbol_in(key)),
            )
                .pushed(span.push)
                .stopped(span.stop)
        })
        .collect();
    project.set_chords(at, end, &markers);
}
```

`let ... else` takes the end of the last span, or returns at once when the chart is empty (`spans.last()` is `None`). `symbol_in(key)` is the same function the Progression menu uses to name “V7” as A7 in D minor, so the sheet spells a demo’s chords exactly as the band reads them. `pushed` and `stopped` take `self` by value and return a changed copy, so they chain.

Any other song, a file saved without chords, an imported MIDI file or recording, gets the chords the app hears in it, when it opens and while its sheet is empty. The button **Write the chords the song plays** does it again on demand, in place of what the sheet says; with a band playing from a sheet that has chords, the button is hidden, since the band’s notes come from the sheet in the first place. How the song is heard, a bar cut into stretches with dynamic programming and each stretch named as a whole, is in [Cutting a bar into chords](#). Two rules make the result read like a chart and not like an analysis. The note after a slash is the lowest note of the whole band, the bass line included, so piano chords voiced C E A over a bass playing A are Am, not Am/C. And a section starts with its chord written again, as a chart writes the chord again at the top of each part:

```
let section_starts = self.playlist.sections.iter().any(|s| s.start == start);
if section_starts || markers.last().is_none_or(|last| last.harmony != harmony) {
    markers.push(ChordMarker::new(start, harmony));
}
```

`is_none_or` is true for an empty list, so the first harmony is always written.

Following the song

Because the markers are on the playlist’s timeline, the tools that move things on the timeline move them too:

- **Copying a section, writing out its repeats** or **putting an intro in front** make room with `Project::make_room`, which shifts every clip, section and marker at or after a step, the chord markers with the rest. Each copy of a section gets its chords, starting with the one the section opens with, even when that chord was written in a bar before it.
- **Putting the song in another time signature** moves each marker with its bar and lands it on the nearest beat of the new bar, since a chord changes on a beat (see [Putting a song in another time signature](#)).
- **Changing the key** moves the sheet the way the notes move, so it keeps naming what plays.

A key change is one method on `HarmonyName`:

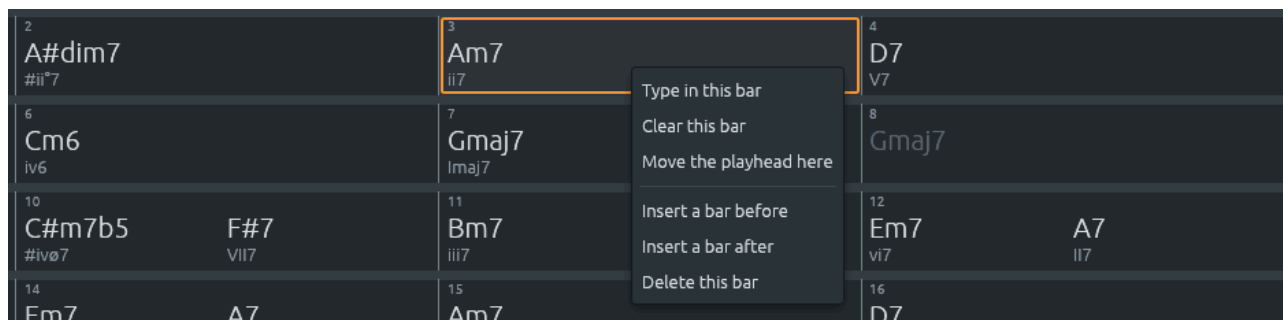
```
match self {
  HarmonyName::Chord(symbol) if from.kind == to.kind => {
    HarmonyName::Chord(symbol.transposed(from.spell(from.root),
    to.spell(to.root)))
  }
  HarmonyName::Chord(symbol) => HarmonyName::Chord(chord_moved(symbol,
from, to)),
  // ...
  HarmonyName::NoChord => HarmonyName::NoChord,
  HarmonyName::Set(set) => HarmonyName::Set(set.transpose(up)),
  HarmonyName::Micromode(set) =>
HarmonyName::Micromode(set.transpose(up)),
  HarmonyName::Row(label) => {
    HarmonyName::Row(RowLabel::new(label.form, label.number + up))
  }
}
```

The first arm has a **match guard**, `if from.kind == to.kind`: it matches a chord only when the mode stays the same. Then the chord moves by letters, as in [Moving a chart to another key](#): G7/B in C major is A7/C# in D major. When the mode changes, the second arm moves each note of the chord to the same degree of the new scale, as the notes in the patterns move: C major in C major becomes C minor in C minor, and G7 becomes Gm7, because its B moves to B $\flat$. Sets and micromodes have no degrees, so they keep their shape and only move with the root; a row form moves with the row.

Inserting and deleting bars

A chart you are writing is rarely right the first time. The bridge needs one more bar before the turnaround, or a transcription has a bar too many. Right-click a bar on the sheet, or a bar number on the playlist's ruler, for **Insert a bar before**, **Insert a bar after** and **Delete this bar** (or **Delete bars 5–8** when several are selected). Each is one step to undo, and the whole song after the place moves along:

```
before:  | C    | F    | G7   | C    |
insert a bar after bar 2:
after:   | C    | F    | %    | G7   | C    |
```



What an inserted bar is

A new bar is **empty**, and empty means “nothing changes here”: the harmony going on before it goes on through it, and so do the key, the tempo and the band’s variation. That is what an empty bar in a chart means too, and it is why the menu doesn’t ask what to put in it.

It is in the **time signature** of the bar it is inserted next to, and in that bar’s **section**. A bar inserted before the first bar of a waltz section is a waltz bar, even though the bar before it is in 4/4: you were looking at the waltz bar when you asked for one more like it. Everything after it moves a bar later: clips, sections, the loop, the time, tempo, key and chord markers, and the band’s variations and section styles.

What a deleted bar takes with it

A deleted bar takes its notes and its chords, and everything after it moves a bar earlier. The question is what happens to **what was going on at its end**. Take C, F, G7, C with the key changing to G major on bar 3 and the tempo to 90 on bar 4, and delete bar 3. The markers in bar 3 go with it, but the key of G major was meant to go on afterwards, and still should. So before anything moves, `closeBars` reads what was going on at the end of the deleted bars:

```
// What goes on at the end of the bars goes on from their start.
let time = meter.time_at(end);
let tempo = self.tempo_map().tempo_at(end);
let key = self.key_at(end);
```

and afterwards sets each of them again where the bars were. Markers that say the same as the one before them are then tidied away, so a delete in the middle of a long stretch in one key doesn’t leave a key marker behind.

The harmony is the subtle one. If a G7 starts in a deleted bar and is held into the next one, the G7 should go on there. But if the bar before the deleted one is a G7 already,

writing another G7 would be a second marker saying the same thing. The code checks both:

```
if let Some(marker) = going_on
    && self.playlist.chord_markers.iter().all(|m| m.start != start)
    && start
        .checked_sub(1)
        .and_then(|step| self.chord_at(step))
        .is_none_or(|before| before.harmony != marker.harmony)
{
    self.playlist
        .chord_markers
        .push(ChordMarker::new(start, marker.harmony));
}
```

This is a **let chain**: `if let` and plain conditions joined with `&&`, stable since Rust 1.88 with edition 2024. Each part can use what an earlier `let` bound: `marker` is used in the last line of the condition. Without let chains this needs an `if let` with a nested `if` inside it. `start.checked_sub(1)` is `None` for the first bar, where there is no “before”; `and_then` passes `None` through and otherwise looks up the chord one step earlier; and `is_none_or` says yes when there is no chord before, or when the one there differs.

Moving every step: one closure

Inserting uses `make_room`, the same method as copying a section. Deleting needs the opposite, and one more case: what was **inside** the deleted steps. One closure says where any step of the song goes:

```
// Where a step of the song goes: the deleted bars close up.
let place = |step: usize| match step {
    step if step <= start => step,
    step if step >= end => step - steps,
    _ => start,
};
```

Steps before the deleted bars stay; steps after them move `steps` earlier; steps inside them land where the bars were. Then it is applied to everything that has a place in the song. A section’s end goes through `place` too, so a section that contained the bars gets shorter, and one made only of them gets a length of 0 and is removed. The loop is kept only if something is left of it:

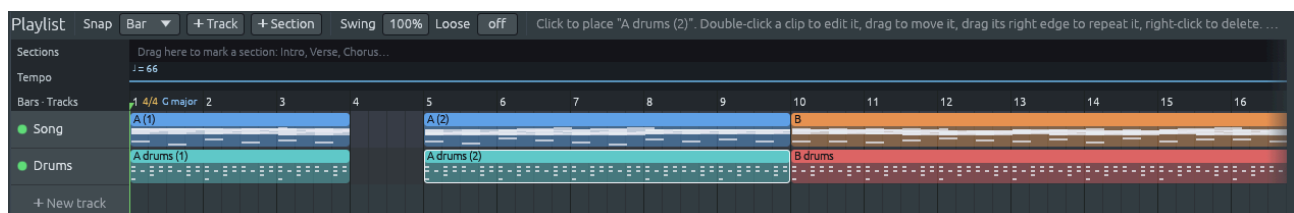
```
playlist.loop_region = playlist.loop_region.and_then(|region| {
    let (from, to) = (place(region.start), place(region.end));
    (from < to).then_some(crate::LoopRegion::new(from, to))
});
```

`bool::then_some` turns `true` into `Some(value)` and `false` into `None`: here “the loop if it still has a length”.

The notes: open the pattern, or cut the clip

A clip plays a **pattern**, and the notes are in the pattern, not in the clip. Inserting a bar in the middle of a clip has to make room in the notes it plays. Two cases:

- **The pattern is played by this clip only, once.** Then the pattern itself can change: it gets the empty steps or loses the deleted ones, and the clip plays the longer or shorter pattern.
- **The pattern is played elsewhere too**, by another clip or by this one repeated. Then the pattern must not change, or the other places would change with it. The clip is **cut in two** where the bar goes in or comes out, and each half plays a pattern of its own: “Riff (1)” and “Riff (2)”.



```
/// Whether clip `index` plays its pattern once and no other clip
/// plays it, so the pattern can change with it.
fn only_clip(&self, index: usize) -> bool {
    let clip = self.playlist.clips[index];
    clip.repeats == 1
        && self
            .playlist
            .clips
            .iter()
            .enumerate()
            .all(|(other, playing)| other == index || playing.pattern !=
clip.pattern)
}
```

`enumerate` pairs each clip with its index, so `all` can skip the clip itself (`other == index`) and ask about every other one. A cut that falls between two repeats of a clip needs no new pattern at all: a clip playing a one-bar groove four times, with its second bar deleted, becomes a clip playing it once and a clip playing it twice.

Notes held across the place are **cut short** there, and the notes after it move:

```
/// `notes` with `steps` empty steps put in at step `at`: the notes from
/// there on move later, and a note held across it ends there.
fn opened(notes: &[NoteEvent], at: usize, steps: usize) -> Vec<NoteEvent>
{
    let at_tick = at * TICKS_PER_STEP;
    notes
        .iter()
        .map(|note| match note.start_tick() < at_tick {
            true => cut_short(note, at_tick),
            false => NoteEvent {
                start: note.start + steps,
                ..*note
            },
        })
        .collect()
}
```

A note held over the new bar is cut where the bar starts, because the new bar is meant to be empty: a note from before sounding under what you write there would be a surprise. `NoteEvent { start: ..., ..*note }` is struct update syntax, a copy of `*note` with `start` changed; it works with `*note` because `NoteEvent` is `Copy`. The halves of a cut pattern keep its colour, and a half cut off only to be deleted is removed again at the end, so a delete leaves no empty patterns behind.

All or nothing

Two things can make an insert or a delete impossible: the song would grow past its longest length, or there is no room for the patterns a cut needs (a song holds at most 999). Either is found out halfway through, after some clips have already changed. So both start by keeping a copy of the project:

```
let before = self.clone();
if !self.open_clips(at, steps) || !self.make_room(at, steps) {
    *self = before;
    return false;
}
```

and put it back if anything fails. `*self = before` replaces the whole project, moving `before` into it. The app then says “There is no room for another bar in the song” or “There is no room for the patterns the cut needs”, and nothing has changed. `||` stops at the first `true`, so `make_room` runs only if `open_clips` succeeded.

The band follows

The band's parts are written from the chord sheet. Moving everything by a bar moves their notes too, and the app must not mistake that for an edit of yours: a part you edit is **frozen**, kept as you left it, and no longer follows the sheet. So both methods run inside `moving_the_band`, which the tools that transpose or re-meter the whole song use too:

```
pub fn insert_bars(&mut self, bar: usize, after: bool, count: usize) ->
bool {
    self.moving_the_band(|project| project.open_bars(bar, after, count))
}
```

`moving_the_band` takes the closure as `impl FnOnce(&mut Project) -> R`, runs it on the project, and then takes the moved parts as the band's own (see [How the band plays a chord sheet](#)).

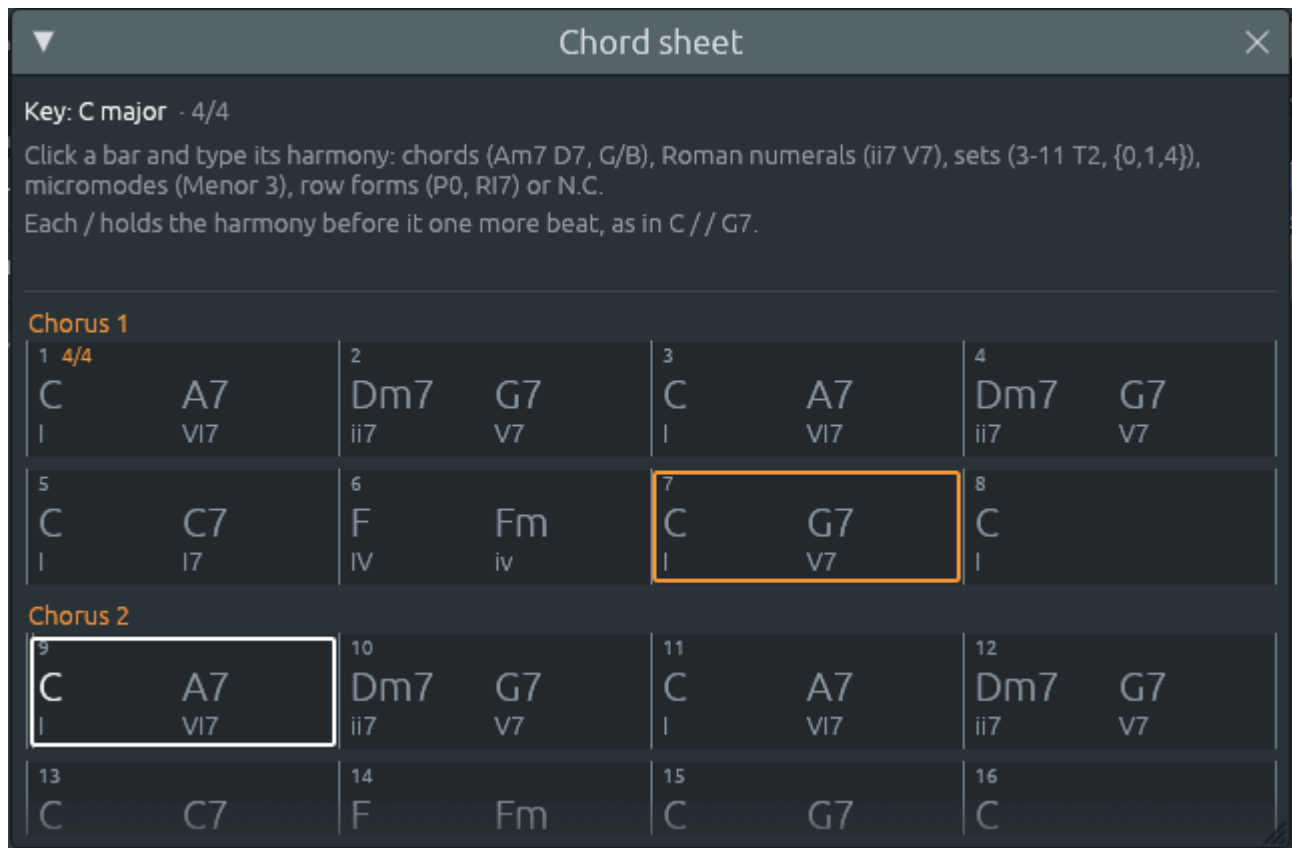
The two menus, on the sheet and on the ruler, are one function, `bar_edits`, so they always offer the same items with the same hints. It returns what was clicked as a value, a `BarEdit`; the sheet wraps it in its own action and the playlist in its own, and the app applies either with `edit_bars`, which also selects the new bar on the sheet so you can type in it at once.

The window

The window is in `lineup-app`, and almost all of it is drawing. Its top rows hold the band's controls, the key, the **Show** menu that names the harmony as chords, sets or micromodes (see [One harmony, four names](#)), the song's form and the practice tools; the chapters on each explain them. Below come the bars.

Each section starts a new line under its name, "Verse ×3" for a section played three times. A double bar line marks where a section starts, and the final bar line (thin, then thick) where the song ends. Each harmony sits where its beat falls, with its Roman numeral under it in the key, its notes under a set or a micromode, and, for a chord from outside the key, what it does there ("V7/ii", "iv · borrowed"; see [What a chord does](#)). An inversion gets the figures of figured bass: G7/B is V⁶₅. A bar with nothing written shows the harmony going on, faintly, so it does not look empty. A key change is written at the top right of its bar, "→ G major". Pointing at a harmony says what it is: "A7: A dominant seventh, V7/ii in C major ... Its notes: A C# E G." Holding the mouse button on it plays it, with its bass note under it.

While the song plays, the bar playing gets a white frame and the chord sounding is the brightest name on the sheet; the others turn dim. The sheet scrolls along with the song.



One rectangle per bar

The sheet is not made of buttons. The window takes the space all the lines need, then works out a `Rect` for every bar (a `Cell`, with its line and column) and asks egui about each one:

```
let id = ui.scope_id().with(("chord-sheet-bar", bar.number));
let response = ui.interact(rect, id, Sense::click_and_drag());
```

`ui.interact` turns any rectangle into something that can be hovered, clicked or dragged, and returns the same `Response` a button would. The id has to stay the same from one frame to the next, because egui remembers by id which widget the mouse went down on; the bar number makes it so. The names, captions and bar lines are then painted with `painter.text`, `painter.vline` and `painter.rect_stroke`.

A text field over a bar

Typing happens in an ordinary `TextEdit`, placed over the bar with `ui.put(rect, field)`. Enter takes the text in and selects the next bar, Tab takes it in and goes on typing in the next bar, Shift+Tab goes back one, and Escape leaves the bar as it was. Three things needed care.

The first letter. You select a bar and type “A”: the field opens in that frame, so the “A” has already been read by the sheet. The field is opened with “A” in it and the cursor after it, by writing the cursor into the field’s saved state before it is drawn. Enter or F2 open a bar with what it holds, all of it selected, so typing replaces it:

```
let mut state = egui::text_edit::TextEditState::load(ui.ctx(),
id).unwrap_or_default();
let range = match fresh {
    Fresh::SelectAll =>
        egui::text::CCursorRange::select_all_str(&editing.text),
    Fresh::CursorAtEnd =>
        egui::text::CCursorRange::one(egui::text::CCursor::new(
            editing.text.chars().count(),
        )),
};
state.cursor.set_char_range(Some(range));
state.store(ui.ctx(), id);
ui.memory_mut(|memory| memory.request_focus(id));
```

The cursor counts characters, not bytes: a “#” typed in a bar takes three bytes in UTF-8, and is one place for the cursor.

Tab and the arrows. egui uses Tab to move between widgets, and a text field gives up the keyboard on it. On the sheet, Tab means “next bar”, so the field takes Tab out of the input before egui sees it:

```
let (tab, back) = if had_focus {
    ui.input_mut(|input| {
        let back = input.consume_key(Modifiers::SHIFT, Key::Tab);
        (input.consume_key(Modifiers::NONE, Key::Tab), back)
    })
} else {
    (false, false)
};
```

and tells egui which keys the sheet keeps while it has the keyboard:

```
const KEEP_KEYS: egui::EventFilter = egui::EventFilter {
    tab: true,
    horizontal_arrows: true,
    vertical_arrows: true,
    escape: false,
};
```

The Enter that closes the field is still “pressed” in that frame, and the sheet’s own keys read Enter as “open this bar”. Without a guard, Enter would close one bar and open the next at once. The sheet remembers whether a bar was being typed in when the frame began, and reads its own keys only if not:

```
let was_editing = self.editing.is_some();
self.edit_field(ui, project, &cells, &keys, sheet.id, actions);
// ...
if has_keyboard && self.editing.is_none() && !was_editing {
    ui.memory_mut(|memory| memory.set_focus_lock_filter(sheet.id,
KEEP_KEYS));
    self.keys(ui, project, &cells, typing_keys, actions);
}
```

While the sheet has the keyboard, digits are part of what you type, “3-11 T2” or “C6/9”, so the harmony keys leave them alone. With the typing keyboard on (letters play notes), a letter cannot start typing in a bar; Enter or a double-click opens it first.

Actions, not changes

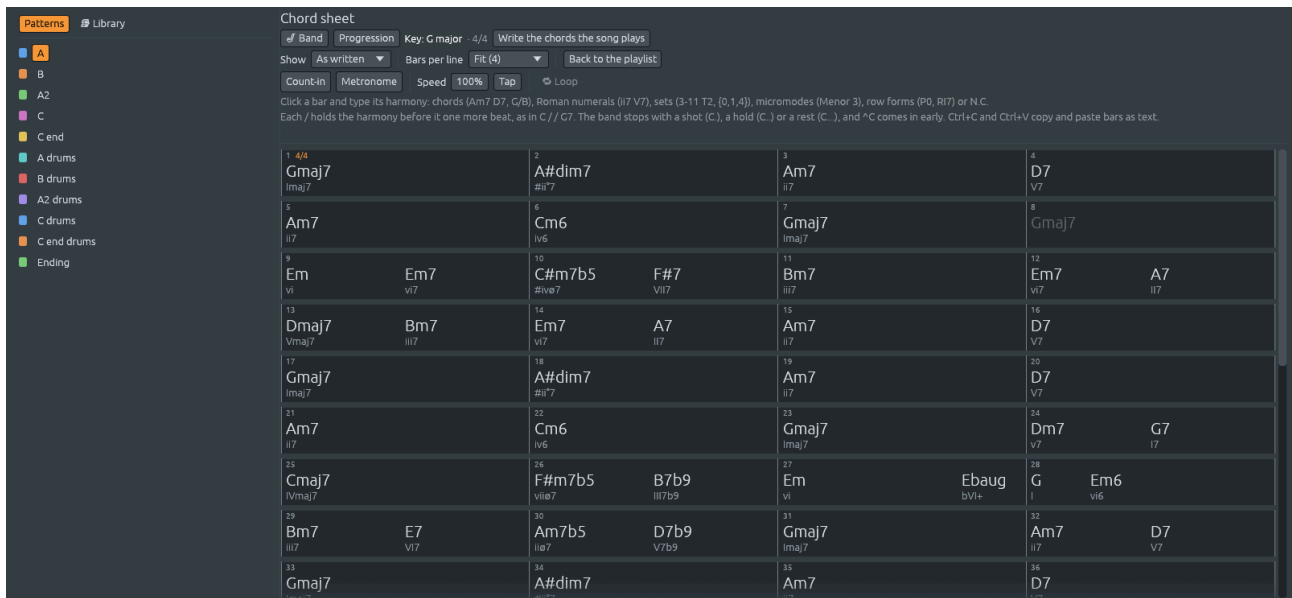
Like the editors of [Part II](#), the sheet never changes the project itself. It draws from `&Project` (a shared, read-only borrow) and returns a list of what was done:

```
pub enum SheetAction {
    // ...
    /// Write the harmonies of the steps from `start` to `end`: one bar.
    SetChords {
        start: usize,
        end: usize,
        markers: Vec<ChordMarker>,
    },
    /// Write the chords the song plays on the empty sheet.
    WriteHeardChords,
    // ...
    /// Insert or delete bars.
    Bars(BarEdit),
    /// Move the playhead to this step of the playlist's timeline.
    Seek(usize),
    /// Play these notes now, as a harmony key does.
    NoteOn(Vec<Note>),
    NoteOff(Vec<Note>),
    // ...
}
```

`LineupApp::apply_sheet` then does each one: `SetChords` through `edit`, so it can be undone, `Seek` by moving the playhead, and the notes through the same path as the harmony keys. The window cannot forget to record an undo step, because it has no way to change the project at all; the borrow checker would refuse it.

Lines that fit the names

A printed chart usually has four bars to a line. The sheet writes much longer names than “C” and “G7”: polychords such as “Eb7|Fb”, sets such as “3-11 T7I”, micromodes such as “Acorde menor-mayor T7I”, each with a numeral or its notes under it. So the number of bars on a line is a choice, **Bars per line**, next to **Show**. By default it is **Fit the names**: as many bars as fit with every name, and the numeral or notes under it, at full size. The menu says how many that is now, as in **Fit (4)**. Or pick a number from 1 to 16; then the line holds that many, and names that don’t fit are written smaller. ☐ **Fill the window** moves the sheet into the main window, in the playlist’s place, where a line has more room.



Why 1, 2, 4, 8, 12 or 16

Fitting could put any number of bars on a line: five, seven, eleven. A printed chart never does, and for a reason. Most songs are made of **phrases of four bars**, and a chart puts a phrase's first bar at the start of a line, so the eye finds bars 1, 5, 9 and 13 in the same column. With five bars to a line, the second phrase would start in the second column of the second line, the third in the third: the song's shape would be lost. So the counts it fits are the ones that keep phrases together: 1 and 2 go evenly into a phrase, and 4, 8, 12 and 16 are made of whole phrases. Twelve is a twelve-bar blues on one line.

```
/// The counts of bars a line holds when they fit, from the most: each
/// divides a four-bar phrase or is made of them.
const FITS: [usize; 6] = [16, 12, 8, 4, 2, 1];

/// No bar is made narrower than this to fit more on a line, however
/// short its names.
const NARROWEST_BAR: f32 = 110.0;
```

`NARROWEST_BAR` is there for songs of short names. A sheet of "C" and "G" would otherwise fit sixteen bars in a window that shows four comfortably, each too narrow to click or to type in.

The choice is a type:

```
#[derive(Debug, Clone, Copy, PartialEq, Eq, Default, Serialize,
Deserialize)]
pub enum Columns {
    /// As many as fit with every name at its full size.
    #[default]
    Fit,
    /// This many, from 1 to [MOST_BARS]: names that don't fit shrink.
    Bars(usize),
}
```

`#[default]` on a variant tells `#[derive(Default)]` which variant `Columns::default()` is. The count comes from one method:

```
pub fn count(self, width: f32, needed: f32) -> usize {
    match self {
        Columns::Bars(bars) => bars.clamp(1, MOST_BARS),
        Columns::Fit => {
            let bar = needed.max(NARROWEST_BAR);
            FITS.into_iter()
                .find(|&bars| width / bars as f32 >= bar)
                .unwrap_or(1)
        }
    }
}
```

`FITS` goes from the most bars to the fewest, so `find` stops at the largest count whose bars are still wide enough. If even one bar is too narrow (a very long name in a very narrow window), it is still one bar to a line, and that bar's names are written smaller.

`FITS.into_iter()` iterates over the array **by value**, so its items are `usize`s. `find` hands each item to its closure by reference, as a `&usize`, so it can test the item without taking it, and give it back if it is the one. The pattern `&bars` in the closure's parameter takes the `usize` back out of the reference.

`count` is a pure function of two numbers, so its test needs no window at all:

```
// Short names: the narrowest bar decides.
assert_eq!(Columns::Fit.count(760.0, 60.0), 4);
assert_eq!(Columns::Fit.count(1300.0, 60.0), 8);
// Long names take more room, and whole phrases still start lines:
// room for three bars makes lines of two.
assert_eq!(Columns::Fit.count(760.0, 240.0), 2);
```

At 1300 pixels, twelve bars would be 108 pixels each, under the narrowest bar; eight are 162. At 760 pixels with names that need 240, three bars would fit (253 each), but three is not a count that keeps phrases together, so it is two.

How wide a bar must be

Before drawing, the sheet asks every bar how wide it must be, and takes the widest. What must fit is each **name** and the **first word under it**: the Roman numeral of a chord, or the notes of a set ("C E G"), which is often the wider of the two. The words after the numeral, such as "borrowed" in "bVII7 · borrowed", are written only where there is room for them. The numeral's colour already says "borrowed", and the hint says all of it; counting those words would let one borrowed chord on the last beat of one bar halve the bars of every line.

```
fn first_word(caption: &str) -> &str {
    caption.split(" · ").next().unwrap_or(caption)
}
```

`split` gives the pieces between the separators, one at a time, and `next` takes the first. It returns an `Option`, because an iterator can be empty; `split` never is, but the type doesn't know that, and `unwrap_or` says what to use if it were.

Each name is placed **on its beat, or just after the name before it** when that one needs the room. In a 4/4 bar of "C A7", C starts on beat 1 and A7 on beat 3. But in a bar where Dm7/C lasts a sixteenth and G7 the rest, G7 is written just after Dm7/C, a little late, as a printed chart does when chords crowd. A name never starts before its beat, so the order of the names is always the order of the chords. A `Row` holds, for each name of a bar, its beat (from 0 at the start of the bar towards 1 at its end) and its width, measured with the painter's fonts (`layout_no_wrap`) the way the names will be drawn:

```
struct Row {
    /// For each name, its beat, from 0 at the start of the bar towards 1
    /// at its end, and its width.
    names: Vec<(f32, f32)>,
}
```

How narrow can the inside of the bar be? Since no name starts before its beat, **each name, with all the names after it, must fit between its beat and the end of the bar**:

```

fn packed_width(&self) -> f32 {
    (0..self.names.len())
        .map(|first| {
            let (beat, _) = self.names[first];
            let from = &self.names[first..];
            let widths: f32 = from.iter().map(|(_, width)| width).sum();
            let gaps = NAME_GAP * (from.len() - 1) as f32;
            (widths + gaps) / (1.0 - beat)
        })
        .fold(0.0, f32::max)
}

```

In “C // A7”, A7 starts on beat 4 and has the last quarter of the bar: an A7 30 pixels wide needs a bar of 120. C starts on beat 1 and needs both names, with the gap between them, in the whole bar. The bar needs the larger of the two. For the sixteenth of Dm7/C, it is the two names side by side, not a bar sixteen times as wide as “Dm7/C”. `&self.names[first..]` is a **slice**: a view of the vector from `first` to its end, without copying it. `sum` has to be told what it adds up to, `f32` here, because many types can be summed from the same items.

The same `Row` places the names when the bar is drawn:

```

fn lefts(&self, left: f32, right: f32) -> Vec<f32> {
    let on_beat = |beat: f32| left + (right - left) * beat;
    let mut free = left;
    let packed: Vec<f32> = self
        .names
        .iter()
        .map(|&(beat, width)| {
            let start = on_beat(beat).max(free);
            free = start + width + NAME_GAP;
            start
        })
        .collect();
    if free - NAME_GAP <= right + 0.5 {
        packed
    } else {
        self.names.iter().map(|&(beat, _)| on_beat(beat)).collect()
    }
}

```

The closure given to `map` changes `free`, a variable outside it, each time it is called: it is an `FnMut`, which `map` accepts because it calls it once per item, in order. When the names don’t all fit that way, on a line of a number of bars you picked, each starts on its beat instead, and they are drawn smaller. Measuring and drawing share one `Row`, so the sheet never makes room for something it doesn’t draw.

The widest bar of the song is a **fold**:

```
.map(|bar| {
    self.needed_width(ui.painter(), project, &meter, &keys, &bar, marks,
&heard)
})
.fold(0.0, f32::max),
```

Why not `.max()`? `Iterator::max` needs `Ord`, a total order, and `f32` doesn't have one: `NaN` is neither smaller nor larger than any number. `f32::max` is a plain function that takes two `f32`s and returns the larger (ignoring a `NaN`), so folding with it from `0.0` gives the widest bar, or 0 for a song with no bars. Passing `f32::max` by name works because a function is a value that can be called, as a closure `|a, b| a.max(b)` would be. Measuring every bar costs a little each frame, but egui caches the text it has laid out recently, so measuring the same name again is a lookup.

With a number picked, a bar can be too narrow for its names. They are then written smaller, with three rules: all the names of a bar at one size, the size the most crowded allows, since a bar that mixes big and small names reads as if the small ones mattered less; a name going on from an earlier bar only shrinks itself; and a set or micromode whose name doesn't fit even at the smallest size is written by its class alone ("Menor 3" for "Menor 3 T8I"), or, where not even that fits, as "...": a name cut short would read as another name.

Taking the playlist's place

The main window has a central panel, where the playlist is drawn. When the sheet fills the window, that panel draws the sheet instead, beside the patterns and the Library:

```

// The chord sheet can take the playlist's place, beside the
// pattern picker and the library.
self.chord_sheet.remember(ctx);
let sheet_fills = self.chord_sheet.fills_window();
// ...
let playlist = egui::CentralPanel::default_margins().show(ui, |ui| {
    if sheet_fills {
        let actions = self
            .playlist
            .browser(ui, &self.project, self.pattern, tried);
        let sheet = self.chord_sheet.main(
            ui,
            &self.project,
            playing,
            typing,
            (practice, self.playlist.library_browser_mut()),
        );
        (actions, sheet)
    } else {
        let actions = self.playlist.show(
            ui,
            &self.project,
            self.pattern,
            song_position,
            tried,
            playlist_zoom,
        );
        (actions, Vec::new())
    }
});

```

Both branches return a pair of lists of actions: what the playlist (or the browser beside the sheet) asks for, and what the sheet asks for. The `if` is an expression, so both branches must have the same type; the playlist's branch returns an empty `Vec` for the sheet's actions. `ChordSheet::main` and the sheet's own window draw the same contents with the same method, `contents`; only the frame around them changes, so everything the sheet does in its window it does in the main window, without a second copy of the code.

`remember` is called **before** `fills_window`. It reads the choices saved by the last run, the first time it is called. Called after, the first frame would draw the playlist and the second the sheet: a flash of the wrong view every time the app starts.

The zoom keys zoom what is under the pointer (see [What is under the pointer?](#)). With the sheet in the playlist's place, the playlist's area must count as nothing, or the keys would zoom a playlist nobody sees:

```
let playlist = match self.chord_sheet.fills_window() {
  true => egui::Rect::NOTHING,
  false => self.playlist.area(),
};
```

`Rect::NOTHING` contains no point at all, so “is the pointer over the playlist?” is always false, and the keys change the interface size. The sheet in the main window keeps the room it has, so smaller names fit more bars to a line. In its own window it does not: the window shrinks with everything else, and holds the same bars.

Remembering the choices

The two choices are one small struct, saved with the app’s state:

```
#[derive(Debug, Clone, Copy, PartialEq, Default, Serialize, Deserialize)]
#[serde(default)]
struct Choices {
  columns: Columns,
  /// Whether the sheet takes the playlist's place in the main window.
  fills: bool,
}
```

`#[serde(default)]` on the struct says: a field missing from what was saved takes its value from `Choices::default()`, so a third choice added later still loads the first two. `remember` keeps a copy of what it last saved, and writes only when the choices differ from it:

```
match self.saved {
  None => {
    self.choices = ctx
      .data_mut(|data| data.get_persisted(choices_id()))
      .unwrap_or_default();
    self.saved = Some(self.choices);
  }
  Some(saved) if saved != self.choices => {
    let choices = self.choices;
    ctx.data_mut(|data| data.insert_persisted(choices_id(), choices));
    self.saved = Some(choices);
  }
  Some(_) => {}
}
```

The `None` arm runs once, on the first frame. The second arm has a match guard: it matches only when the pattern does and the condition holds. Everything else falls to `Some(_) => {}`, nothing to do, which is almost every frame.

The arrows land in the right column

However many bars a line holds, the up and down arrows go to the bar above or below. They never count bars: the sheet keeps every bar it drew as a cell with its **line** and its **column**, and the down arrow looks for the cell on the next line whose column is closest:

```
let in_line = |line: usize, column: usize| {  
  cells  
    .iter()  
    .filter(|cell| cell.line == line)  
    .min_by_key(|cell| cell.column.abs_diff(column))  
    .map(|cell| cell.bar.number)  
};
```

The cells come from what was drawn, so they are right with four bars to a line, with sixteen, and on a short line at the end of a section. `min_by_key` works here, unlike `max` above, because the key is a `usize`, which is `Ord`.

Chord sheets as text

Musicians send each other chord charts as plain text: in an email before a rehearsal, in a forum post, in a notes app. Nobody agreed on a format, but most charts look alike, because they copy the way a chart is printed:

```
Title: Autumn Song  
Key: C major  
Time: 4/4  
Tempo: 120  
Style: Bossa nova  
  
[Verse x2]  
| C      | Am7 | Dm7 G7 | C  |  
| F      | G7  | G7     | C  |  
  
[Chorus]  
| F      | G   | Em7 A7 | Dm |
```

File > Import chord sheet... opens such a file (or one dropped on the window) as a new song with a band: the chart's key, time signature, tempo and style, its sections on the playlist, its chords on the sheet. If something can't be read, nothing is imported, and the status bar says where: "line 4: bar 3: "Qx7" is not a chord ...". **File > Export chord sheet...** writes any song's sheet in the same format, with the bars lined up in columns. In the sheet, **Ctrl+C** copies the bars selected as text (or the whole sheet, with

its settings and sections, when all or none are selected), **Ctrl+X** cuts them, and **Ctrl+V** writes the text on the clipboard from the selected bar on. One line without any `|` is read as one bar, so “Dm7 G7” copied from a message goes straight into a bar. A paste takes only the chords: the song keeps its own key, time signature, tempo, band and sections.



The format

Everything is optional except the bars.

Line	Means
Title: Autumn Leaves	The song's name. A first line that is nothing else is the title too, as at the top of a printed chart.
Key: G major	The key: G, Em, E minor, A Dorian, C harmonic minor and the other tonal scales (not rows or sets, which are chosen in the app). A key keeps the name written: Gb major stays Gb major.
Time: 3/4	The time signature, any the app takes: 6/8, 5/4, 2+2+3/8.
Tempo: 132	Quarter notes per minute; 132 BPM reads too.
Style: Bossa nova	The band's style, by its name.

Line	Means
[Verse]	A section starts with the next bar. [Chorus x2] , [Chorus x2] or [Chorus] x2 plays it twice.
C Am7 Dm7 G7	Bars. A line can hold any number of them.
# Ask Ana about the bridge	A note for people, skipped.

Inside a bar goes exactly what you type in a bar of the sheet, marks included. || , |: and :| read as plain bar lines; repeats are written with x2 on a section, which is how the playlist stores them. A Key: or Time: line between bars changes the key or the time signature from the next bar on. Roman numerals are read in the chart's key, so

```
Key: F major
| I | vi | ii7 V7 | I |
```

becomes F, Dm, Gm7 C7, F, and the same text with Key: D major gives D, Bm, Em7 A7, D: one progression in any key. With no style: line, the band plays a style in the chart's time signature, and with no time signature either, the style a new song starts with.

An empty bar reads as on the sheet: the harmony before goes on through it. %, the simile sign, plays the bar before again, as a musician reads it: after a bar of two chords, "| C G | % |" is C G twice, and after a bar of one chord it is the same as an empty bar.

read_bar sees only its own bar, so the sign is read by read_typed_bar_after , which is also given the bar before and its markers. repeat_bar copies those markers into the new bar, each as many steps into it, with its shot, hold, rest or push. A first marker that would only go on with the harmony already sounding, as in "| C | % |", is left out, so that bar stores exactly what an empty one does:


```

pub fn repeat_bar(before: &[ChordMarker], before_bar: &Bar, bar: &Bar) ->
Vec<ChordMarker> {
    let in_bar: Vec<&ChordMarker> = before
        .iter()
        .filter(|marker|
(before_bar.start..before_bar.end()).contains(&marker.start))
        .collect();
    let sounding = in_bar.last().map(|marker| marker.harmony);
    let mut markers: Vec<ChordMarker> = in_bar
        .iter()
        .map(|marker| ChordMarker {
            start: bar.start + (marker.start - before_bar.start),
            ..**marker
        })
        .filter(|marker| marker.start < bar.end())
        .collect();
    if let Some(first) = markers.first()
        && first.start == bar.start
        && Some(first.harmony) == sounding
        && first.push.is_none()
        && first.stop.is_none()
    {
        markers.remove(0);
    }
    markers
}

```

The song stores the chords, not the text: once read, the bar holds a C and a G of its own, and clicking it shows “C G”.

Reading the text

`text::read` turns the text into a `TextSheet`, and only checks the layout. It can’t read the bars yet: how many beats a bar has depends on the time signature at that point, which only the song knows once the `Time:` lines before it are in place.

```

pub struct TextSheet {
    pub title: Option<String>,
    /// The key the song starts in.
    pub key: Option<Scale>,
    /// The time signature the song starts in.
    pub time: Option<TimeSignature>,
    /// In quarter notes per minute.
    pub tempo: Option<f32>,
    /// The name of the band style to play it in, as written, and its
    /// line.
    pub style: Option<(String, usize)>,
    /// The bars, sections and changes, in order.
    pub items: Vec<Item>,
}

pub enum Item {
    /// A bar, as it is typed in the chord sheet: "Dm7 G7".
    Bar { text: String, line: usize },
    /// A section starts with the next bar.
    Section {
        name: String,
        repeats: usize,
        line: usize,
    },
    /// The key changes from the next bar on.
    Key(Scale),
    /// The time signature changes from the next bar on.
    Time(TimeSignature),
}

```

`Item` is an enum whose variants hold different things: two have named fields, like a struct, and two hold one value, like a tuple. A `match` on an `Item` has to say what it does with each of the four, so when a fifth kind of line is added, the compiler lists every place that must learn about it. Bars and sections keep the number of their line, so a problem found later, when the bar is read, can still name it. The lines are numbered with `enumerate`, which counts from 0:

```

for (index, line) in text.lines().enumerate() {
    let number = index + 1;
    let line = line.trim();
    if line.is_empty() || line.starts_with('#') {
        continue;
    }
}

```

The error is a small struct with the line and the message, and `impl fmt::Display` says how it prints, "line 4: ...". Every function that can fail returns `Result<_, TextError>`, and `?` passes the error up at once. `map_err` turns another error into this one on the way:

```
let key = read_key(value).map_err(|message| error(number, message));
```

A line is a setting when it has a colon and the word before it is one of the five settings, whatever its case. `split_once(':')` splits “Key: G major” at the first colon into `("Key", " G major")`, and returns `None` when there is no colon:

```
fn setting(line: &str) -> Option<(String, &str)> {
    let (name, value) = line.split_once(':')?;
    let name = name.trim().to_lowercase();
    ["title", "key", "time", "tempo", "style"]
        .contains(&name.as_str())
        .then_some((name, value))
}
```

A section heading can put its count inside the brackets or after them. `read_times` reads “x2”, “x2” or “X 2”:

```
fn read_times(text: &str) -> Option<Option<usize>> {
    let number = text
        .strip_prefix(['x', 'x', 'X'])?
        .trim()
        .parse::<usize>()
        .ok()?;
    Some((1..=MAX_REPEATS).contains(&number).then_some(number))
}
```

`strip_prefix` takes an array of characters and strips whichever one the text starts with. The answer is an option inside an option, because there are three cases to tell apart: `None` is “this is not a count at all” (the heading goes on with other words), `Some(None)` is “a count, but too many times” (an error), and `Some(Some(2))` is twice. Where the heading is read, a let chain looks for the count at the end of the name:

```
if repeats.is_none()
    && let Some((first, last)) = name.rsplit_once(char::is_whitespace)
    && let Some(times) = read_times(last)
{
    repeats = Some(times);
    name = first.trim().to_string();
}
```

Making the song

`text::import` starts from `Project::with_band()`, the song **File > New** makes, and sets what the chart says, in the order the band needs it: first the style (which can put

the empty song in its own time signature), then the chart's time signature, key and tempo on top. Then it walks the items once, keeping the step where the next bar starts:

```
Item::Bar { text, line } => {
    if step + time.steps_per_bar() > MAX_SONG_STEPS {
        return Err(error(*line, tr("the song cannot be this long.")));
    }
    bars.push((step, text, *line));
    step += time.steps_per_bar();
}
Item::Time(new) => {
    time = *new;
    project.set_time_marker(TimeMarker::new(step, time));
}
Item::Key(key) => {
    project.set_key_marker(KeyMarker::new(step, *key));
}
Item::Section { name, repeats, .. } => sections.push((name.clone(),
*repeats, step)),
```

A section ends where the next one starts, or at the end of the song. Once every marker is in place, each bar is read with `read_typed_bar`, which gets the bar (to know its beats) and the key at its start (to read numerals). All the chords go into the song in one `set_chords`, and the band is told to play the sheet. `text::paste` does the same reading into an existing song, from a bar on, and returns the markers instead of writing them; the window turns them into one `SheetAction::SetChords`, the same action as typing in a bar, so undo and the band need nothing new.

Writing the text

`text::write` goes through the bars with `write_bar`, the function that fills a bar's text field when you click it, and collects lines: words ("[Verse ×2]", "Key: D minor") or a row of up to four bars. A section, a change of key or of time signature starts a new row, and an empty bar is written `%` when playing the bar before again sounds the same, as after "`| C |`". After "`| C G |`" a held bar is written `/ / / /`, a `/` for each beat, since `%` would read back as C G again. It writes in letters and English, whatever the app shows, so the text reads the same wherever it is pasted:

```
pub fn write(project: &Project, title: Option<&str>) -> String {
    Language::English.with(|| NoteNaming::Letters.with(||
write_sheet(project, title)))
}
```

Then `layout` lines the bars up. It finds the widest bar in each of the four columns, then pads every bar to its column's width:

```
for (width, bar) in widths.iter_mut().zip(bars) {
    *width = (*width).max(bar.chars().count());
}
```

```
.map(|(bar, width)| format!("{bar:<width$}"))
```

`{bar:<width$}` writes `bar` left-aligned (`<`) in a field as wide as the variable `width` (the `$` says the width comes from a variable). The widths are counted with `chars().count()`, not `len()`: `len()` counts the bytes of the UTF-8 text, and a letter outside plain ASCII (ø, x, b) takes two or three bytes but one place on the screen. A key the text can't say, a twelve-tone row or a pitch-class set, is written as a note for people, “# Key: ... (chosen in the app)”, so the chart still reads and nobody is misled.

Copy, cut and paste

egui turns Ctrl+C, Ctrl+X and Ctrl+V into events of their own, with the pasted text already taken from the system clipboard. That is the same on Linux, macOS (where the keys are Cmd+C, Cmd+X and Cmd+V) and Windows. The sheet looks for them among the frame's events while it has the keyboard and no bar is being typed in; while a bar is being typed in, they go to the text field as usual.

```
for event in &input.events {
    match event {
        egui::Event::Copy => copy = true,
        egui::Event::Cut => cut = true,
        egui::Event::Paste(text) => paste = Some(text.clone()),
        _ => {}
    }
}
```

To copy, `ctx.copy_text(text)` puts the text on the clipboard at the end of the frame. A cut is a copy followed by a `SetChords` with no markers over the bars selected. With no bar selected, a copy takes the whole sheet, but a cut would clear nothing, so it leaves the clipboard alone.

Testing it

The best test of a pair of functions like `write` and `read` is a **round trip**: write something, read it back, and check you got the same thing.

`every_song_demo_comes_back_from_its_text` does that for every song demo: each one's chord sheet is written as text, imported, and its chords, keys and time signatures compared with the demo's. That test is why "C G7 C /" keeps its slash and why a name never runs across a /. A few demos have a chord between two beats, which a chord sheet can't write, so the test compares the chords placed on the beats, and checks that fewer than one demo in ten has such a chord. Next to it,

`a_bar_is_written_back_as_it_reads` and

`every_name_is_written_back_the_same_way` do the same for one bar and one name.

In `chord_sheet.rs`, `harmonies_spread_over_the_beats_as_on_a_lead_sheet` checks the table above, `a_bar_cannot_hold_more_than_its_beats` the refusal,

`every_way_of_naming_harmony_is_read` and `nonsense_is_refused_with_a_reason` the names, `numerals_are_chords_in_the_key` and

`typed_numerals_in_minor_follow_the_way_numerals_are_written` the numerals, and `harmonies_move_with_the_key` a change of key. In `text.rs`,

`a_chart_is_read_line_by_line`, `bar_lines_are_read_however_they_are_drawn`, `a_mistake_says_its_line`, `numerals_are_read_in_the_charts_key`,

`pasted_bars_go_from_the_bar_chosen_on` and `the_sheet_is_written_in_columns` cover the text.

The tests in `bars.rs` work on a small song: one pattern of four bars playing C, F, G7, C, a verse over all of it, a loop over bars 2 and 3, the key of G from bar 3 and 90 BPM from bar 4. `an_inserted_bar_moves_everything_after_it` checks every note, chord, key, tempo, the verse and the loop;

`a_bar_inserted_before_a_change_of_time_takes_the_new_time`,

`a_clip_of_a_pattern_played_elsewhere_too_is_cut_in_two`,

`a_deleted_bar_closes_up`, `a_deleted_bar_leaves_no_pattern_behind`,

`a_harmony_from_a_deleted_bar_goes_on_where_it_was` and

`a_repeated_clip_loses_the_repeats_in_the_deleted_bars` follow the cases above, and `the_band_follows_the_bars` checks that no part of the band was frozen. To hear that nothing else moved, the [playback trace](#) of a demo before and after an insert shows every note before the bar at the same time, and every note after it exactly one bar later.

In the app, `lines_fit_as_many_whole_phrases_as_there_is_room_for` checks `count`.

`lines_hold_as_many_bars_as_show_every_name_in_full` starts a song whose first bar is "Dm7/C F/C G7/D C/E" and checks that every name is drawn at its full size, 20 points, then picks 8 bars and finds "Dm7/C" smaller.

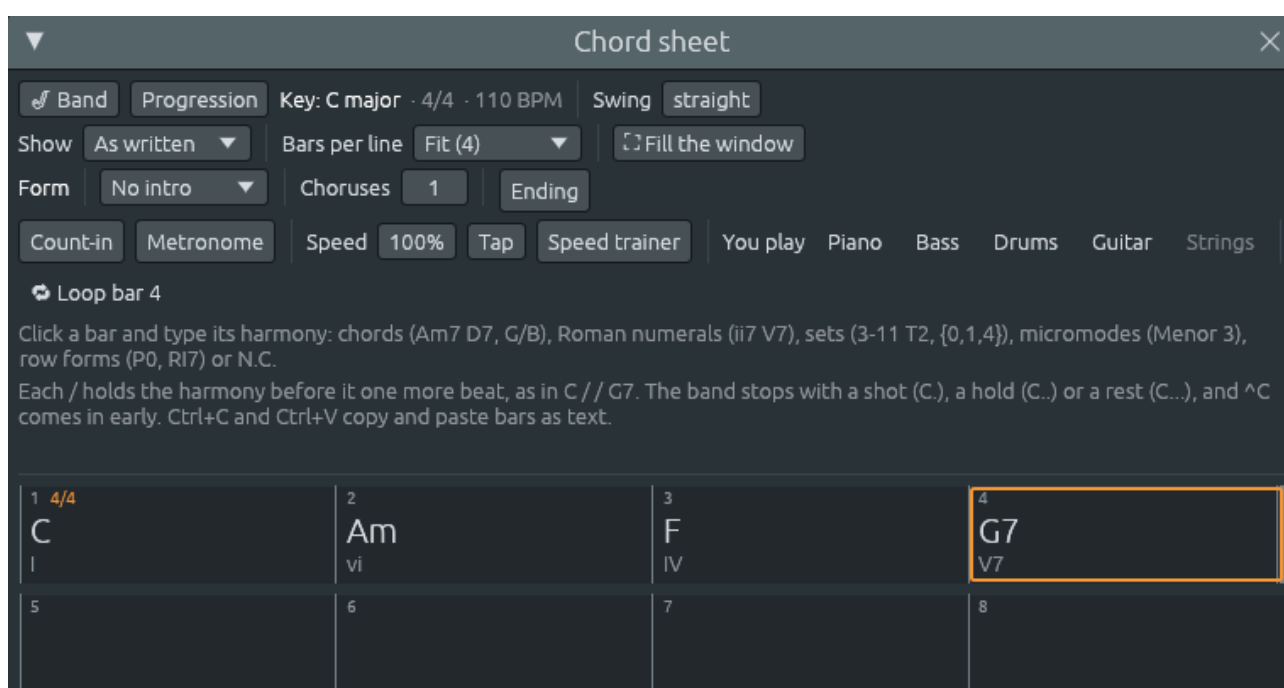
the_words_after_a_numeral_are_written_where_there_is_room and
a_short_chord_keeps_its_name_whole_with_the_next_just_after_it check the
placing, the_zoom_keys_over_the_sheet_in_the_main_window_zoom_the_sheet and
the_sheet_fills_the_main_window_and_goes_back the main window, and
aBars_menu_inserts_a_bar_and_deletes_it and
the_bar_numbers_insert_and_delete_bars_too the two menus.

Exercises on the chord sheet are in [Exercises: the band](#).

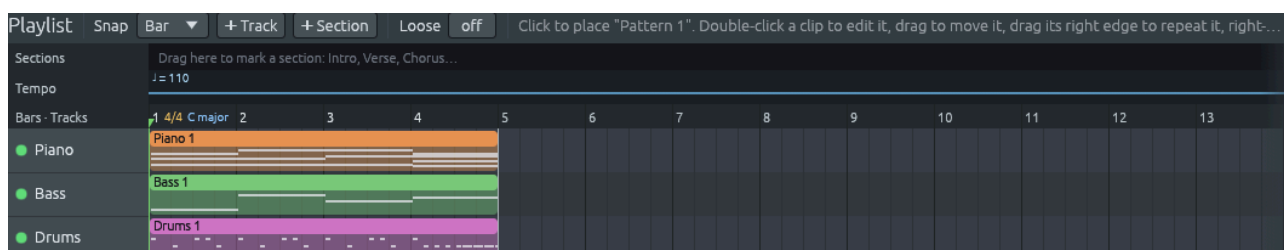
How the band plays a chord sheet

Files: `crates/lineup-core/src/band.rs`, `crates/lineup-core/src/band/`, `crates/lineup-core/src/library/comping.rs`, `crates/lineup-core/src/library/follow.rs`, `crates/lineup-app/src/chord_sheet/band.rs` and `crates/lineup-app/src/app/follow.rs`

The chord sheet holds a song's harmony. The band plays it. Type `C`, `Tab`, `Am`, `Tab`, `Dm7`, `G7`, `Tab`, `C`, press Space, and a piano, a bass and drums play those chords, with a guitar, strings, an organ or a horn section where the style has them. Change the style, the key or a chord, and the band plays it again the new way.



The band's parts are ordinary patterns on tracks of their own, called "Piano 1", "Bass 1", "Drums 1". Double-click one and the piano roll shows every note the band plays; nothing is hidden, which matters in a program you learn from: you can see how a bossa nova pianist spreads a Dm7 over the bar, or where a walking bass puts its lead-in note.



This chapter covers how a style is split from the chords it plays, what each player plays and how, how the parts are written again when the sheet changes without losing what you edited, the bigger band (guitar, strings, organ, horns), and the arranger-keyboard mode where the band follows the chord you hold. Song form, variations and section styles are in [Song form, sections, variations and locks](#); breaks in the band's playing in [Shots, holds, rests and pushes](#). How to use the band is in the manual's [Your first song](#) and [A bigger band and the horns](#).

The music, briefly

A few words come up all through the band's code:

- **Comp**: a chord played under the tune; as a verb, to play those chords. **Comping** is the rhythm and the voicings a chord player uses, such as a bossa pianist's syncopated chords.
- **Voicing**: which notes of a chord sound, and in which octaves. **Voice leading** moves each voice of a chord to the nearest voice of the next, so the chords join smoothly.
- **Changes**: a song's chord progression, bar by bar; what the chord sheet holds.
- **Walking bass**: one note per beat, the root first, notes of the chord, and a half step into the next root.
- **Pad**: chords held softly under the band, usually by strings or an organ.
- **Turnaround**: the chords at the end of a section that lead back to its start, such as I vi ii V.

What to play, and how to play it

A style from the [library](#) is a bundle: "Bossa nova" means Dm7 G7 Cmaj7 *and* the piano's syncopated chords *and* the bass on 1 and the "and" of 2 *and* the rim clicks. A backing band separates the two halves. The chords are yours, on the sheet; the style only says how to play them.

Half	What it is	Where it is chosen
What to play	a progression of numerals	the sheet's Progression menu, from each <code>style</code> 's chart
How to play it	comping, bass, drums, the players it adds	the Library , as a <code>BandStyle</code>

A `BandStyle` is a style with the chords taken out:

```
pub struct BandStyle {
    pub name: &'static str,
    pub genre: Genre,
    pub time: TimeSignature,
    /// The tempo it is usually played at, in BPM.
    pub tempo: f32,
    pub comping: Comping,
    pub bass: Bass,
    /// The drums, or `None` for a band without them.
    pub rhythm: Option<&'static Rhythm>,
    pub about: &'static str,
}
```

The band styles are not written out by hand. Each library rhythm already knows how the piano and bass go with it (`apply::feel` , see [Playing a song in another style](#)), so the list is made from the rhythms the first time it is asked for, with four styles without drums added: **Chorale** (each chord held, as an organ plays a hymn), **Alberti bass** (the chord broken low, high, middle, high, as in Mozart's sonatas), **Arpeggios** and **Sarabande** in 3/4.

```
pub fn band_styles() -> &'static [BandStyle] {
    static STYLES: OnceLock<Vec<BandStyle>> = OnceLock::new();
    STYLES.get_or_init(|| {
        let mut styles: Vec<BandStyle> = RHYTHMS
            .iter()
            .map(|rhythm| {
                let (comping, bass) = apply::feel(rhythm);
                BandStyle {
                    name: rhythm.name,
                    genre: rhythm.genre,
                    time: rhythm.time,
                    tempo: rhythm.tempo,
                    comping,
                    bass,
                    rhythm: Some(rhythm),
                    about: rhythm.about,
                }
            })
            .chain(WITHOUT_DRUMS)
            .collect();
        // A stable sort: within a genre, the library's order stays.
        styles.sort_by_key(|style| style.genre);
        styles
    })
}
```

A `static` must be built when the program is compiled, and a `vec` cannot be, because it needs memory from the heap. `std::sync::OnceLock` is a box that starts empty and is filled once, the first time `get_or_init` runs; every later call gets the

same value back. Several threads can race to call it and only one closure runs. The return type `&'static [BandStyle]` says “this list lives as long as the program”, which is what lets a `&'static BandStyle` sit in the project as the band’s style with no lifetime to track. `sort_by_key` is stable, so styles of one genre keep the library’s order.

A band in the song

File > New makes a song with an empty sheet and a band ready to play it in **Pop beat**:

```
pub fn with_band() -> Self {
    let mut project = Project::empty();
    project.set_band_style(BandStyle::named(DEFAULT_STYLE));
    project.remove_unused_tracks();
    project
}
```

From there, three places shape the band:

- **The Library.** A click on a style or a rhythm makes it the band’s way of playing, with the parts ticked in its window: the drums, the chord and bass parts, the voices, the chord progression, the time signature, the tempo and the swing (see [Which parts to take](#)). A song with music of its own but no band gets one when the chord and bass parts are taken with the voices.
- **The sheet’s 🎵 Band menu** chooses who plays and on which sound: an **Instrument lineup** (“Piano, acoustic guitar + strings”, “Horns, timpani + strings” and six more), a tick box and an instrument for each player, the **Horn section**, the **Soloist** (see [The soloist](#)) and **No band**.
- **The sheet’s Progression menu** is the Library again, showing only styles with a chord progression; a click writes that progression from the selected bar on, in the song’s key: I vi ii V in C is Cmaj7 Am7 Dm7 G7, the turnaround that ends so many standards. A chord already sounding is not written again, so “I | I | IV” writes C, nothing, then F, as a chart does. On an empty sheet, the song takes the progression’s mode: “i bVII bVI” is Am G F in A minor.

Because the parts can come from different places, the band keeps them apart:

```

pub struct Band {
    pub custom_rhythm: Option<Box<crate::library::UserStyle>>,
    pub custom_instruments: Option<Box<crate::library::UserStyle>>,
    pub style: &'static BandStyle,
    /// The instrumentation can come from a different style than the
    rhythm.
    pub instrument_style: &'static BandStyle,
    /// The exact library style supplying the comping, when one was
    picked.
    pub harmony_style: Option<&'static Style>,
    /// The chords and the bass played as plain accents on the beats: how
    /// songs saved so play. The Library no longer chooses it.
    pub accents_only: bool,
    /// Whose drums play, when not those of `style` or `custom_rhythm`.
    pub drums: BandDrums,
    // ...
}

pub enum BandDrums {
    /// Those of the style the band plays the chords and the bass in, or
    /// of the saved style it plays them from.
    Style,
    /// Those of another style.
    Of(&'static BandStyle),
    /// Those of a saved style.
    Saved(Box<crate::library::UserStyle>),
    /// None: the song's own drums play.
    Song,
}

```

`style` says how the chords and the bass are played; `drums` says whose drums play with them; `instrument_style` says who plays and on what; and `harmony_style` is the library style picked, when its comping differs from its rhythm's. The two `custom_` fields hold styles of your own (see [Your own styles](#)). `accents_only` plays the chords and the bass on the beats the bar is felt in, whatever the style; only songs saved that way still play so.

Taking a style's chord and bass parts but not its drums keeps the drums the band played: `drums` turns from `style` into `of` the old style, so they no longer follow `style`. Taking the drums alone sets `of` the new one. When the two end up the same again, `tidy_drums` turns `of` back into `style`, so a band has one way to say "this style's drums", and its file stays one an older version can read. Each player asks the band, not the style, how to play:

```

fn comping_in(&self, variation: Variation) -> Comping {
    if self.accents_only {
        Comping::Accents
    } else {
        self.playing_style().comping_in(variation)
    }
}

```

Picking a style can change three more things. A style in another time signature puts the song in it, bar for bar, as a rhythm from the library does (see [Putting a song in another time signature](#)): each bar keeps its length in seconds, so a song in 4/4 at 110 BPM picked as a Jazz waltz becomes 3/4 at 82.5 BPM ($4 \times 60 / 110 = 3 \times 60 / 82.5 = 2.18$ seconds a bar). The tempo is the style's while the sheet is empty, and yours once you have written chords. And the song takes the style's swing, since swing is part of how a style is played. The piano and the bass get the General MIDI instrument of the genre (an electric piano for soul and electronic styles, a double bass for jazz and tango, a synth bass for hip hop), so they sound right through a SoundFont or MIDI out.

What each player plays

A band style knows *how*; the sheet knows *what*. Putting them together takes three steps.

The harmonies, as notes

`played_harmonies` walks the sheet and turns each harmony into a `Played`: when it starts and ends, the notes the chords are played with (its voicing), and what the bass plays under it.

```

pub struct Played {
    pub start: usize,
    pub end: usize,
    /// The notes the chords are played with, from low to high. Empty for
    /// no chord: nothing plays.
    pub voicing: Vec<Note>,
    /// What the bass plays.
    pub bass: BassPart,
}

/// What the bass plays under a harmony.
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub enum BassPart {
    /// A chord: its line goes over the chord's notes from `note` (its
    /// root, or a slash chord's bass), and may lead into `next`, the
    /// bass note of the harmony after it.
    Chord {
        chord: Chord,
        note: Note,
        next: Note,
    },
    /// A group of notes with no root, such as a pitch-class set: the
    /// bass keeps the rhythm of its line on this one note and its octave.
    Pedal(Note),
    /// Nothing.
    Rest,
}

```

Chords are voiced the way the library voices a chart, each from the one before, moving every voice as little as it can, so C Am F G7 does not jump around the keyboard (see [Voice leading](#)). Their bass notes make a line (`bass_line`), so G7/B has B in the bass. The other ways Ferrous Lineup names harmony are played by their notes:

- **A polychord** is played as written: the lower chord, the upper stacked right above it, over the lower chord's bass.
- **A pitch-class set or a micromode** has no root, so there is no "right" bass note. The band takes the first note of its normal form (its most compact order): 3-11 T2, D F A, is played from middle C up, over a D. The choice is arbitrary but fixed, so the same set always sounds the same.
- **A row form** is a line of twelve notes, not a chord. The band plays it as four chords of three notes, in the row's order, one after another, each over its first note. Cutting a row into groups and sounding each group as a chord is a common way to harmonise it; Webern built the row of his Concerto, Op. 24, from four such groups.
- **N.C.** plays nothing; the drums go on.

The rhythm, kept to the bars

The library's comping patterns start again at every chord. That is right for a chart where chords change on the bar line, but a sheet can say `Dm7 G7` (a change on beat 3) or `C / / G7` (on beat 4). Starting the pattern again on beat 3 would shift the groove every time the chord changes, as if the drummer counted "one" in the middle of the bar. A band keeps its groove and lets the chords fall into it. So

`Comping::play_in_bars` places the rhythm's hits in every bar, from its first beat, and each hit plays whichever harmony is sounding when it falls:

```
for bar in bars {
  let (period, hits) = self.hits(bar.time);
  let from = if self.follows_the_bar() {
    bar.start
  } else {
    period_start(bar, period)
  };
  for offset in (0..bar.end() - from).step_by(period) {
    for hit in hits.iter() {
      let at = from + offset + hit.at;
      if at < bar.start || at >= bar.end() {
        continue;
      }
      let index = harmonies.partition_point(|harmony| harmony.start
<= at);
      let Some(index) = index.checked_sub(1) else {
        continue;
      };
      // ...
    }
  }
}
```

The harmonies are sorted by start, so `partition_point` counts, by binary search, the harmonies that have started by `at`; the sounding one is the one before. A chord so short that no hit falls in it (a G7 on the "and" of 4, in a pattern that only strikes on 1 and 3) would go unheard, and a skipped chord is a wrong harmony, so such a chord is struck once where it starts. A pattern longer than a bar, like the bossa nova's two-bar clave, counts from bar 1: `period_start` starts it in odd bars and goes on in even ones, the way a player counts "one, two" across the pair.

Beats for any meter

Most comping patterns are a fixed table of hits. Two are not: **plain accents** (`Comping::Accents`), a chord on each beat the bar is felt in, and the **aksak** of the Balkan styles, the afterbeats inside each beat. In 7/8 felt as 2+2+3 they fall in one

place, in 5/4 felt as 2+3 in another, so their hits are built from the bar's `felt_groups` (see [Beats of different lengths](#)):

```
fn hits(self, time: TimeSignature) -> (usize, Cow<'static, [Hit<Voices>]>)>
{
    match self {
        // ...
        Comping::Aksak => {
            let unit = time.unit_steps();
            // A chord a little shorter than its note, so the next
            // one is heard to start, but never under a step.
            let length = (unit - unit / 4).max(1);
            let mut hits = Vec::new();
            let mut at = 0;
            for group in time.felt_groups() {
                for note in 1..usize::from(group) {
                    let velocity = if note == 1 { 84 } else { 74 };
                    hits.push(hit!(at + note * unit, length, Voices::All,
velocity));
                }
                at += usize::from(group) * unit;
            }
            (time.steps_per_bar(), Cow::Owned(hits))
        }
        _ => {
            let (period, hits) = self.pattern();
            (period, Cow::Borrowed(hits))
        }
    }
}
```

The function returns a `Cow`, “clone on write” (see [The order of everything](#)): the fixed patterns hand out a borrow of their `static` table, `Cow::Borrowed`, at no cost, and the two built ones an owned `Vec`, `Cow::Owned`. The caller reads both the same way. These two patterns count from the bar line wherever the chord starts (`follows_the_bar`), since the beats of an odd meter are where the bar puts them.

The bass

`Bass::play_in_bars` is the same idea for the bass line: it starts again where each harmony starts, so the root lands on the change, and again at each bar line, so a chord held for eight bars does not turn into one long walk. Only the last piece of a harmony leads into the next. A walking bass plays one note per beat:


```

fn walk(span: usize, down: bool) -> Vec<(usize, Tone)> {
    const UP: [Tone; 6] = [
        Tone::Third,
        Tone::Fifth,
        Tone::Seventh,
        Tone::Octave,
        Tone::Fifth,
        Tone::Third,
    ];
    const DOWN: [Tone; 6] = [
        Tone::LowSeventh,
        Tone::LowFifth,
        Tone::LowThird,
        Tone::LowFifth,
        Tone::LowSeventh,
        Tone::Root,
    ];
    let beats = span.div_ceil(4);
    (0..beats)
        .map(|beat| {
            let tone = if beat == 0 {
                Tone::Root
            } else if beat == beats - 1 {
                Tone::Approach
            } else if beats == 3 {
                if down { Tone::LowFifth } else { Tone::Fifth }
            } else if down {
                DOWN[(beat - 1) % DOWN.len()]
            } else {
                UP[(beat - 1) % UP.len()]
            };
            (beat * 4, tone)
        })
        .collect()
}

```

The root on 1, chord notes in the middle, and on the last beat the **approach**, a note a half step above or below the next chord's bass note. The line is given that note itself, not just its name: a root is played between C2 and B2 (see [The harmonies, as notes](#)), so from G7 to C the next note is C2, and the approach is the half step on the side nearer the note before it, D $\flat$ 2 coming down from D2. Knowing only "C", the line could pick B2 and then fall a seventh to the C2 that follows.

The middle notes go up the chord (**UP**) or down it (**DOWN** , the chord's notes below the root). `Bass::play` makes both lines and keeps the one that reaches the next chord in smaller steps, the line down only while it stays above E1 (`BASS_FLOOR`), the lowest string of a bass guitar. So over C going to Am the walk goes up, C2 E2 G2 G $\sharp$ 2, into A2, and over G7 going to C it comes down, G2 F2 D2 D $\flat$ 2, into C2. In a bar of three the walk goes root, fifth, approach: over C7 going to F, C2 G2 F $\sharp$ 2, into F2. The third is left

out there because the third of C is the half step below F: the line would reach its lead-in a beat early.

The drums

The drums play the style's rhythm in every bar, with its fill in the last bar of every eight (`PHRASE_BARS` , as most songs are built), in the last bar before a section or a change of variation, and at the end of the intro and of the chorus. A change of variation or style starts with a crash cymbal. Variations, the intro and the ending are in [Song form, sections, variations and locks](#).

Each part, as patterns

A whole song does not fit in one pattern (a pattern holds at most 256 steps, 16 bars of 4/4), and one giant pattern would be hard to read anyway. `chunks` cuts the song into stretches: whole bars of one time signature, inside one section, as long as a pattern may be. `write_part` cuts the notes at those lines (a note that lasts past the end of a stretch goes on at the start of the next) and compares each stretch's notes with those before:

```
let index = match contents.iter().position(|known| *known == content) {
    Some(index) => index,
    None => {
        contents.push(content);
        contents.len() - 1
    }
};
```

Stretches that play exactly the same notes share **one** pattern, with two clips of it, as a producer copies a clip instead of playing a part twice. A song of two identical choruses gets "Piano 1" twice in the playlist. The patterns are written over the ones the player already had, so "Piano 1" keeps its number while you edit the sheet, and a piano roll open on it shows the new notes instead of closing. Every pattern the band writes is tagged:

```
/// The band's player whose part it holds, if the band wrote it (see
/// [crate::band`]). The band writes these patterns again when the
/// chord sheet changes.
pub player: Option<Player>,
```

That tag is how the band finds its own work again, and how the library knows to leave it alone when it plays a whole song in another rhythm.

Playing again when the sheet changes

The band plays what the sheet says *now*. Type a chord, change the key, put the song in another meter, drag a section: the parts are written again, in the same undo step as your change, so one Ctrl+Z takes back both. Every change to the project goes through

`LineupApp::edit` (see [Undo and redo](#)), so that is where the band is told:

```
let before = self.project.clone();
let result = change(&mut self.project);
// The band plays what the chord sheet says now, in the same step.
self.project.follow_chord_sheet();
```

Writing every part again after every click would be wasteful: moving a clip of the melody changes nothing the band plays from. So `follow_chord_sheet` first takes a **fingerprint** of what the parts are played from (the style, the chord markers, the keys, the bars, the sections, the form) and compares it with the one taken last time:

```
pub fn follow_chord_sheet(&mut self) -> bool {
    let Some(band) = &self.band else {
        return false;
    };
    if band.heard == Some(self.band_inputs()) {
        return false;
    }
    self.band_channels(false);
    self.write_band();
    true
}
```

A fingerprint, or hash, is a number computed from some data, such that different data almost surely give different numbers. Comparing two 64-bit numbers is much cheaper than comparing two chord sheets, and the band only has to keep the number, not a copy of the sheet.

A part you edit stays yours

A band that writes its parts afresh every time would lose an edit to the bass line the next time anything changed. Here, a part you change by hand **freezes**: the band leaves it as you left it, and the sheet says so, with a **Play it again** button that throws your version away and has the band write the part from the sheet once more. The other players go on following the sheet.



How does the band know you edited a part, whatever tool you used: the piano roll, the channel rack, a dragged clip, the library? It does not watch the tools. It keeps, for each player, a fingerprint of the part it last wrote, and compares it with a fingerprint of the part as it is now:

```
pub fn is_frozen(&self, player: Player) -> bool {
    self.band.as_ref().is_some_and(|band| {
        band.written[player.index()].is_some_and(|written| written !=
self.part_print(player))
    })
}
```

Two `is_some_and` nested read as one sentence: “there is a band, and it has written this player’s part, and that is not what is there now”. `part_print` mixes in where each of the player’s clips starts, how often it repeats, and every note of its pattern. Which track a clip is on and which number its pattern has are left out, so moving the Piano track down, or deleting a pattern of yours (which renumbers the ones after it), freezes nothing.

FNV-1a

The fingerprints are saved in the project file, so a part you froze stays frozen when you open the song tomorrow. The standard library’s `DefaultHasher` is not promised to give the same number in the next version of Rust, so it cannot be saved. The band uses **FNV-1a**, a classic hash of a few lines:

```

struct Fingerprint(u64);

impl Fingerprint {
    fn new() -> Self {
        Fingerprint(0xCBf2_9CE4_8422_2325)
    }

    fn bytes(&mut self, bytes: &[u8]) {
        for &byte in bytes {
            self.0 ^= u64::from(byte);
            self.0 = self.0.wrapping_mul(0x0100_0000_01B3);
        }
    }

    fn number(&mut self, number: usize) {
        self.bytes(&(number as u64).to_le_bytes());
    }
}

```

For each byte: mix it into the number with XOR, then multiply by a large prime, `0x1000000001B3`. The starting number and the prime are the published constants of the algorithm, so every program that implements FNV-1a gets the same result. Multiplying two large `u64`s overflows; in a debug build plain `*` would stop the program with “attempt to multiply with overflow”, and `wrapping_mul` says that keeping only the low 64 bits is what we want, as a hash does on purpose. `to_le_bytes` writes a number’s bytes lowest first (“little-endian”), whatever the computer, so the same song gives the same print everywhere. A part with no notes at all is the starting number itself, 14695981039346656037: you will see it in a saved file, for strings that wait for variation b.

Moving the whole song is not an edit

Changing the song’s key transposes every note, the band’s included. The band’s parts no longer match their fingerprints, but you did not edit them: the tool moved everything. Putting the song in another meter, copying a section and inserting a bar are the same. These tools run inside `moving_the_band`:

```

pub(crate) fn moving_the_band<R>(&mut self, change: impl FnOnce(&mut
Project) -> R) -> R {
    self.keeping_parts(|_, _| true, change)
}

```

```

fn keeping_parts<R>(
    &mut self,
    keep: impl Fn(&Project, Player) -> bool,
    change: impl FnOnce(&mut Project) -> R,
) -> R {
    let following: Vec<Player> = Player::ALL
        .into_iter()
        .filter(|&player| {
            self.band
                .as_ref()
                .is_some_and(|band|
                    band.written[player.index()].is_some())
                && !self.is_frozen(player)
                && keep(self, player)
        })
        .collect();
    let result = change(self);
    let prints: Vec<(Player, u64)> = following
        .into_iter()
        .map(|player| (player, self.part_print(player)))
        .collect();
    if let Some(band) = &mut self.band {
        for (player, print) in prints {
            band.written[player.index()] = Some(print);
        }
    }
    result
}

```

It notes which parts were following the sheet before, runs the tool, and then takes their new fingerprints as the band's own. A frozen part is moved by the tool like any other notes, and stays frozen: your version of the piano, now in D. `keep` is an `impl Fn`, called many times, while `change` is an `impl FnOnce`, called once. The prints are collected into a `Vec` before `self.band` is borrowed mutably, because `part_print` needs `self` too, and the borrow checker allows one or the other at a time. Deleting a channel uses the same method with another `keep`: a part that played nothing on the deleted channel only had its channel numbers shift, which is not an edit; a part that lost notes is frozen.

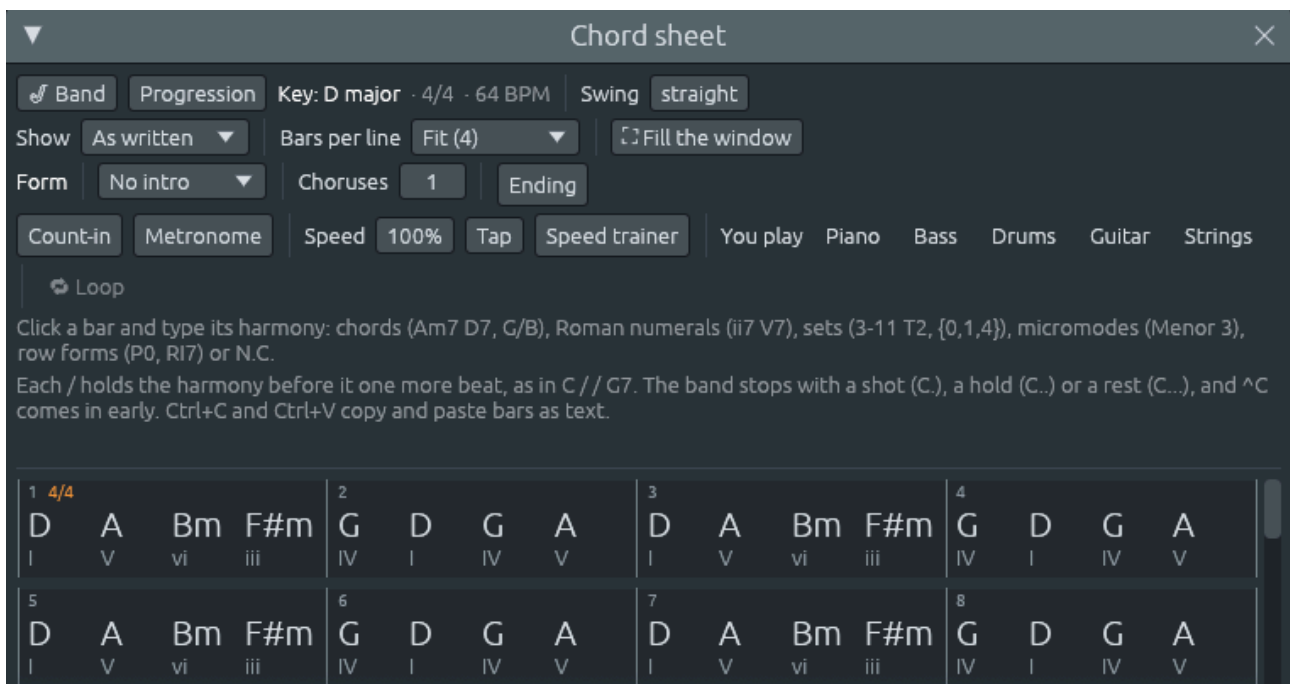
A band over a song of its own

Open a demo, such as **Canon in D**, and take a style's chord and bass parts and its voices from the Library. A demo already plays its own accompaniment, so first the band:

1. writes the chords the song plays on the sheet, if it is empty (see [Where the chords come from](#)), and
2. **mutes** the song's own chord and bass channels (the library calls them "Keys" and "Bass") and, when the style's drums come too, its tracks that hold only drums,

so the two do not play over each other. The melody plays on, now with the band behind it. Without the style's drums, the band comes in with **drums:**

BandDrums::Song, and the song's own drums play on. A style that follows the song, such as the orchestra, comes in alongside the song's parts instead of muting them (see [An orchestra for music that changes meter](#)).



No band removes the band's patterns, its tracks and its channels (only the ones it added, and only if nothing else plays on them), and plays again what it muted. It remembers what it muted by *name*, not by number, so the right channels come back even after you added or removed others in between. The chords stay on the sheet.

A bigger band: guitar, strings and organ

A piano, a bass and drums is a jazz trio, and the rhythm section of most bands, but a rock band without its guitar or a tango without its violins does not sound like itself. So a style can bring up to three more players, each playing the way that music uses them:

Style	Who comes in	What they play
Pop beat	steel-string guitar, strings in b	strummed chords, held strings

Style	Who comes in	What they play
Rock beat	overdriven guitar, rock organ in b	power chords in eighth notes
Shuffle, Rock and roll	clean electric guitar	the boogie shuffle
Swing	jazz guitar	four to the bar
Funk, Disco	muted guitar, and strings in disco	sixteenths, scratched
One drop, Steppers, Rocksteady	clean guitar, organ	the chop on 2 and 4, the organ bubble
Bossa nova	nylon-string guitar	the bossa rhythm
Tangos	strings	held chords
Chorale	church organ	held chords
Hip hop, most electronic styles	nobody	

“Strings in b” means they wait for variation b: right-click a bar, pick **Variation b**, and from there the strings come in, holding the chords an octave above the piano, as strings lift a pop chorus.

The music: how a guitarist plays a chord

A pianist can play any notes of a chord: ten fingers, every key in reach. A guitarist has six strings, tuned from the lowest

String	6	5	4	3	2	1
Open note	E2	A2	D3	G3	B3	E4
MIDI	40	45	50	55	59	64

and a left hand with four fingers to press them against the **frets**, one semitone each. Each string plays one note at a time, or none. A string not pressed rings **open**, for free; that is why the keys of E, A, D and G, whose chords use many open strings, are the guitar’s home keys. So a guitar chord is a **shape**: a fret (or nothing) for each string. Chord books write it as six characters from the low E string, with **x** for a string not played and **0** for an open one:

C	x32010	the A string at fret 3 (C), D at 2 (E), G open (G), B at 1 (C), high E open (E): C E G C E
G	320003	
Am	x02210	
F	133211	the first finger flat across fret 1: a "barre"

The lowest string that sounds must be the chord's **bass**, the note the bass player plays (the root, or under a slash chord the note after the slash). That is why the C shape leaves out the low E string: an E there would make it C/E, which chord books write 032010 .

Ferrous Lineup does not store a chord book. It finds the shapes the way a guitarist learns them, by trying the frets in reach and keeping the easiest. For each **position** on the neck (the fret the first finger sits at), it tries every way to put a note of the chord on each string within four frets, keeps the ones a hand can play, and scores them:

- every **needed** note must be there: the root, the third (or the second or fourth of a sus chord), the sixth or seventh, a fifth that is not perfect, and the highest extension, which names the chord (the 13 of a G13);
- a perfect fifth and the lower extensions are **welcome**, and shapes with more of them are better;
- shapes low on the neck, with **open strings**, and on more strings are better;
- up to four pressed strings are fine. More need a **barre**, one finger flat across the lowest fret, which stops every string above it, so none of those can be open, and leaves three fingers for the rest.

One more rule keeps the sound clear. Two notes close together sound muddy low down: a third between E2 and G#2 rumbles, while a fifth there is fine. Arrangers have a table of the lowest note each interval sounds clear from, the **low interval limits**, and a shape that goes below them costs a little more. A little, not a lot: the open G, 320003 , starts with G2 and B2, a third below its limit (Bb2), and it is still the G every guitarist plays. Its sound is part of the guitar's sound.

With those rules, the search finds the open chords of the chord books (C x32010 , G 320003 , D xx0232 , Am x02210 , G7 320001 , Cmaj7 x32000), barre chords in other keys (F 133211 , the open E shape a fret higher, and B x24442 , the A shape), and a shape for every chord type in every key.

A guitarist does not strike the strings together. The pick sweeps across them: on a **down-stroke** from the low strings to the high ones, on an **up-stroke** back from the high ones, and the hand coming up rarely reaches the low strings. The library's rhythms already mark the light "up" hits, between the beats. The guitar plays each down-stroke on every string, low to high, a tick apart (about 5 ms at 120 BPM, so a six-

string strum takes about 25 ms, as a real one does), and each up-stroke on the top four, high to low. How swing moves a strum as a whole is in [Swing](#).

What each style's guitarist plays:

- **Strummed chords** (pop, folk, country): the full shape in the style's rhythm.
- **Power chords** (rock, punk, metal): only the root, its fifth and the octave, **A5** = A2 E3 A3. Distortion adds new notes at the sums and differences of the frequencies played. For a fifth, whose frequencies are close to 3:2, they fall in line with the root, an octave or two below it; for a third they land on notes out of tune with the chord, and it buzzes and sours. So rock guitarists leave the third to the bass and the singer, and "A5" means "A without its third".
- **Four to the bar** (swing): three-note chords, the root, the third and the seventh, struck on every beat, the second and fourth a little harder, as Freddie Green kept time in Count Basie's band for almost fifty years. The shapes are the jazz **shells**: from a root on the low E string, the seventh comes next and the third an octave up (G7 = G2 F3 B3); from the A string, the third and then the seventh (C7 = C3 E3 B♭3).
- **The boogie shuffle** (blues, rock and roll): two strings only, the root with its fifth on one beat and with its sixth on the next, both eighths of each beat: A and E, A and E, A and F♯, A and F♯. Over a minor chord it stays on the fifth.
- **The chop** (reggae, soul, Motown, bluegrass): a short chord on the top four strings on beats 2 and 4, with the snare, the low strings left to the bass. Country calls it the "chick" of "boom-chick". In **ska** the guitar plays between all the beats instead.
- **Scratching** (funk, disco): sixteenth notes on a muted guitar, some accented, most only a percussive "chk".

Strings hold each chord for as long as it lasts, an octave above the piano, softly: a pad. Ballads use the slower-starting strings, synth styles the synthesizer ones. The **organ** holds chords too, as a church organ in a chorale or a Hammond in a slow blues. In reggae it plays the **bubble**: short chords between all the beats low down, and on beats 2 and 4 an octave higher, with the guitar's chop.

Every player plays a General MIDI instrument chosen by the style. The General MIDI table numbers them from 1, the MIDI messages from 0, so the code says 24 for the nylon guitar:

Program (from 0)	Instrument	Styles
24	Acoustic guitar (nylon)	bossa nova, samba, bolero, folklore, rumba flamenca

Program (from 0)	Instrument	Styles
25	Acoustic guitar (steel)	pop, folk, country, bluegrass, waltz
26	Electric guitar (jazz)	swing
27	Electric guitar (clean)	shuffle, rock and roll, surf, Motown, reggae, ska
28	Electric guitar (muted)	funk, disco
29, 30	Overdriven, distortion guitar	rock, punk, metal
48, 49, 50	Strings, slow strings, synth strings	pop, tango, ballads, R&B, trance
16, 17, 18, 19	Drawbar, percussive, rock, church organ	blues, reggae, rock, chorale

Players in arrays

The band keeps one channel and one fingerprint per player, in arrays indexed by the player:

```
pub const COUNT: usize = 8;

pub const ALL: [Player; Player::COUNT] = [
    Player::Piano,
    Player::Bass,
    Player::Drums,
    Player::Guitar,
    Player::Strings,
    Player::Organ,
    Player::Soloist,
    Player::Horns,
];

pub(crate) channels: [Option<usize>; Player::COUNT],
```

`band.channels[player.index()]` answers “which channel?” for every player, and `index` is `self as usize`: a field-less enum converts to its position. The array’s length is part of its type, `[Option<usize>; 8]`, so the compiler checks that `ALL` really lists `COUNT` players. `write_band` loops over `Player::ALL`, and everything after it, the patterns, the tracks and the fingerprints that keep an edited part frozen, works for every player the same way.

Who plays in which style: a table of constants

Each style's added players are a value of this type:

```
pub struct Extra<Part> {  
    pub part: Part,  
    pub program: u8,  
    pub only_in_b: bool,  
}
```

```
pub struct Extras {  
    pub guitar: Option<Extra<GuitarPart>>,  
    pub strings: Option<Extra<PadPart>>,  
    pub organ: Option<Extra<PadPart>>,  
}
```

`Extra` is **generic**: `Part` is a type parameter, filled in as `GuitarPart` for the guitar and `PadPart` for strings and organ. The three fields “what it plays, on which instrument, does it wait for b” are written once for both. The styles are listed in a table built by the compiler:

```
const NONE: Extras = Extras {  
    guitar: None,  
    strings: None,  
    organ: None,  
};  
  
const fn with_guitar(part: GuitarPart, program: u8) -> Extras {  
    Extras {  
        guitar: guitar(part, program),  
        ..NONE  
    }  
}
```

```
const BY_STYLE: &[amp;(&str, Extras)] = &[  
    // Pop: a strummed acoustic guitar, strings for the chorus.  
    (  
        "Pop beat",  
        Extras {  
            guitar: guitar(STRUMMED, STEEL_GUITAR),  
            strings: pad_in_b(STRINGS),  
            ..NONE  
        },  
    ),  
    // ...  
];
```

A `const fn` can be called where Rust needs a value at compile time, such as in a `const ..NONE` is struct update syntax: the fields not written take their values from `NONE`, so each style names only the players it has. The test `every_style_listed_is_a_band_style` checks that every name in the table is a real band style, so a typo cannot quietly leave a style without its guitar.

Enums that carry data, and a function as a parameter

What the guitar plays is an enum whose variants can carry a value:

```
pub enum GuitarPart {
    Strum(Comping),
    StylesRhythm,
    Chop,
    Power(Comping),
    FourToTheBar,
    Boogie,
}
```

`Strum(Comping::Bossa)` and `Strum(Comping::Waltz)` are one variant with different rhythms inside. A `match` can look inside it, with `|` patterns nested, as in variation b, where held and picked chords become strummed ones:

```
pub fn in_variation(self, variation: Variation) -> GuitarPart {
    match (variation, self) {
        (Variation::B, GuitarPart::Power(Comping::Held)) =>
            GuitarPart::Power(Comping::Pulse),
        (
            Variation::B,
            GuitarPart::Strum(Comping::Held | Comping::ArpUp |
            Comping::ArpUpDown),
        ) => GuitarPart::Strum(Comping::Strum),
        (_, part) => part,
    }
}
```

Strummed chords, power chords and shells differ only in which notes each harmony gets; playing the rhythm and strumming it is the same. So `strummed` takes the voicing as a parameter:

```
fn strummed(
    rhythm: Comping,
    bars: &[Bar],
    harmonies: &[Played],
    voice: fn(&Played) -> Vec<Note>,
) -> Vec<NoteEvent> {
```

`fn(&Played) -> Vec<Note>` is a **function pointer** type: any plain function with that signature fits, and the caller passes `shape`, `power_chord`, `shell` or `top_strings` by name. Finding a shape tries thousands of fret combinations, and a song repeats its chords, so `strummed` remembers the shapes it found in a `HashMap`:

```
let voicing = shapes.entry(key).or_insert_with(|| voice(harmony)).clone();
```

`entry` finds the place for `key` in the map, and `or_insert_with` calls the closure only if that place is empty: each different chord is searched once.

Trying every fret: recursion with a callback

How many strings a shape uses is not known in advance, so the search cannot be a fixed number of nested `for` loops. It is a function that picks a fret for one string and calls itself for the next:

```
fn try_frets(
    choices: &[Vec<u8>],
    low: usize,
    top: usize,
    frets: &mut Vec<u8>,
    found: &mut impl FnMut(&[u8]),
) {
    let string = low + frets.len();
    if string > top {
        found(frets);
        return;
    }
    for &fret in &choices[string] {
        frets.push(fret);
        try_frets(choices, low, top, frets, found);
        frets.pop();
    }
}
```

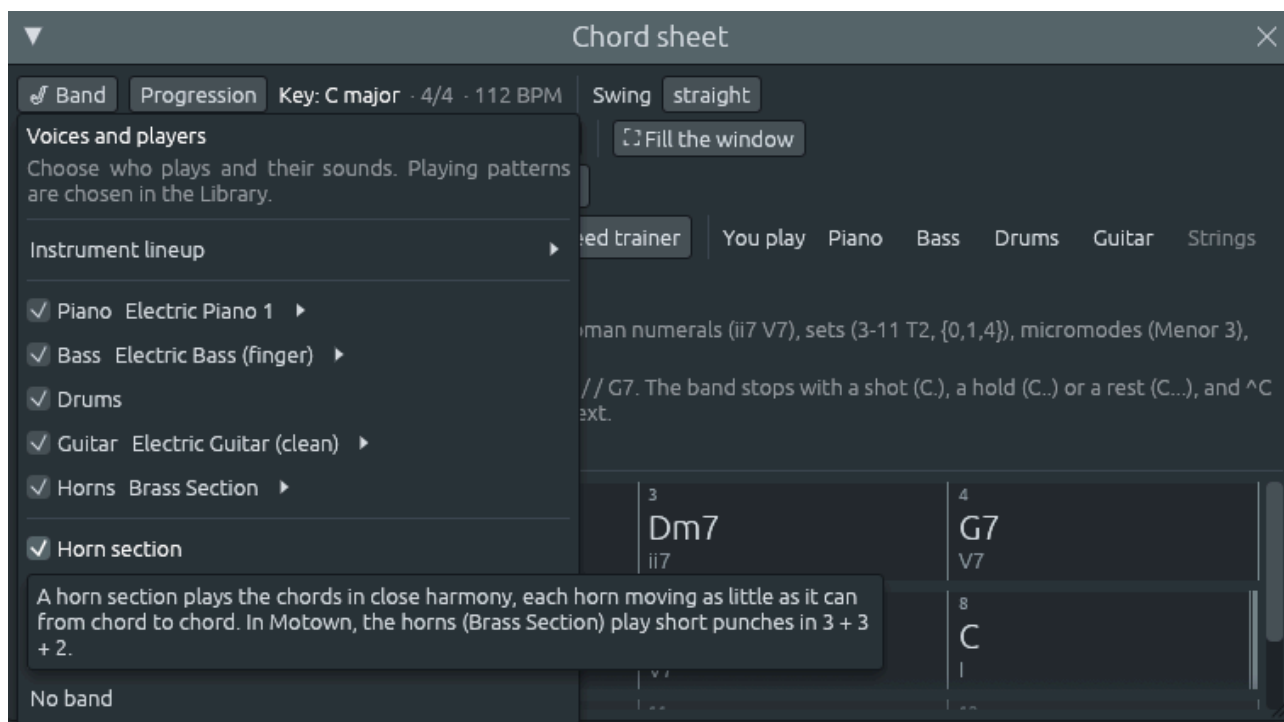
`frets` is one `Vec` shared by every level: a level pushes its string's fret, lets the deeper levels finish, and pops it again before trying the next. When every string has a fret, `found` gets the whole shape. `found` is a closure that scores it and keeps the best one, so it changes something outside itself: that makes it an `FnMut`, and it is passed as `&mut` so that every level of the recursion calls the same closure, rather than each taking its own.

A horn section

Soul and funk records have one. So do big bands, salsa orchestras, ska bands and the brass bands of the Balkans: trumpets, saxophones and trombones playing the chords together. Tick **Horn section** in the Band menu, and a **Horns** track comes in with the band, playing the chord sheet in a part that suits the style:

Style	What the horns play
Ballads (Pop ballad, Jazz ballad, Slow blues, Neo soul, R&B, Bossa nova, Bolero...)	long notes held under the band
Swing, shuffle, blues, rock and roll	big-band kicks on beat 1 and just before beat 3 (the Charleston rhythm)
Pop, rock, soul, hip hop, Latin, second line	short punches in 3 + 3 + 2 (the tresillo)
Funk	tight funk stabs in sixteenth notes
House and other electronic styles	syncopated stabs, as in house music
Reggaeton, moombahton	short chords in the dembow's 3 + 3 + 2
Ska	short chords between the beats
March, waltz, polka, Balkan dances	afterbeats, the "pah" of a brass band's oom-pah
Tango, folk, classical, reggae, styles without drums	long notes
Other styles in 3/4 or 6/8	long notes
Other styles in 5/4, 7/8 and other odd meters	a punch on each beat the bar is felt in

The **3 + 3 + 2** rhythm splits the eight eighth notes of a 4/4 bar into groups of three, three and two, with an accent at the start of each: on beat 1, on the "and" of 2, and on beat 4. Played this way, the bar pushes forward. It is the tresillo of Cuban music, and it is everywhere in pop and soul horn lines. The **afterbeats** are the off-beats of a brass band: in a march in 2/4 the tuba plays "oom" on the beat and the horns "pah" between; in a waltz the bass has beat 1 and the horns beats 2 and 3. The instrument is **Brass Section** (General MIDI 61) for most styles, **French Horn** for classical ones and **Synth Brass** for electronic ones.

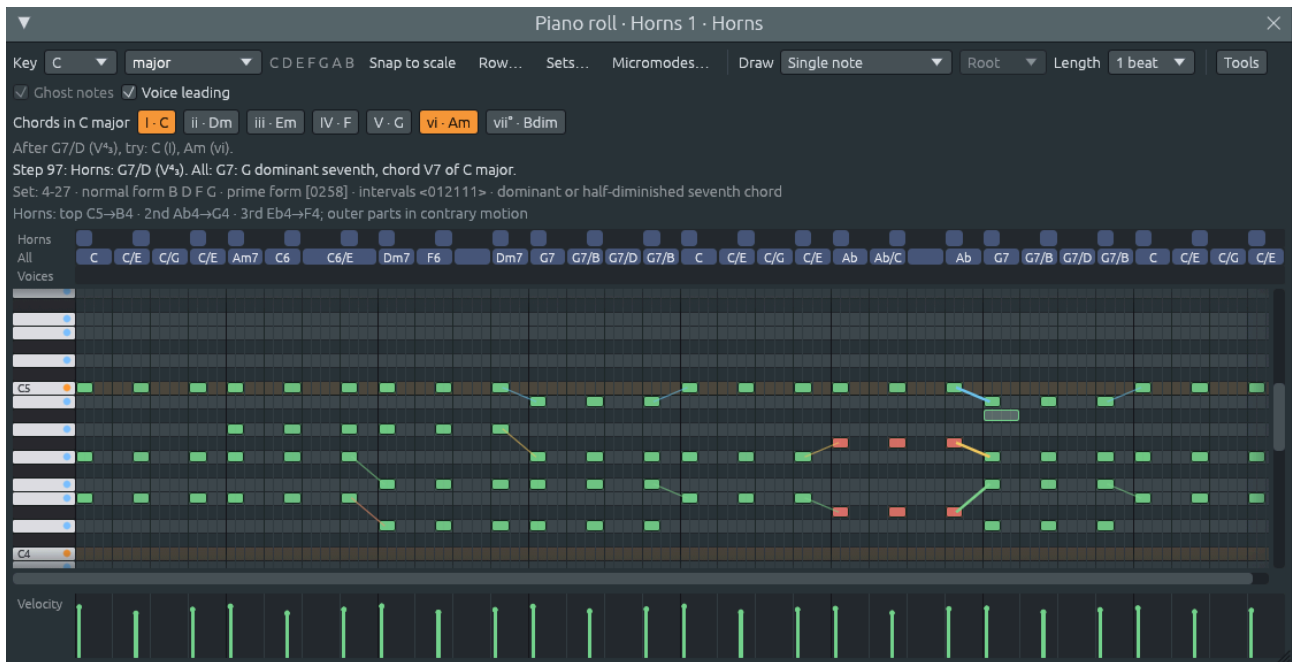


Close harmony, moving as little as possible

Horns playing chords together voice them in **close position**: the notes of the chord as near together as they go, inside an octave, under the highest horn, the **lead**.

Arrangers write horn parts that way because it makes three or four horns sound like one big instrument. The horns play the notes the piano plays in the chord, whatever their octave: as many horns as the piano's voicing has different notes, up to four. A chord with more than four notes, such as C13 (C E G B $\flat$ D A), leaves out its fifth first, and then its root, which the bass plays anyway: the horns play E, B $\flat$, D, A, what makes the chord sound as it does. The lead stays between F4 and F5, where a trumpet section sounds bright without straining.

From one chord to the next, the horns take the close voicing that moves least: each horn goes to the nearest note it can. C to A $\flat$ is E4 G4 C5 to E $\flat$ 4 A $\flat$ 4 C5: two horns move a semitone, down and up, and the lead holds C. A $\flat$ to G7: the lead falls from C5 to B4, the second horn from A $\flat$ 4 to G4, the third rises from E $\flat$ 4 to F4, and a fourth comes in on D4 for the seventh chord. Then from G7 to C, B4 rises to C5 and F4 falls to E4, as the leading tone and the seventh of a dominant do.



The horns also avoid **parallel fifths** where they can. Two horns a perfect fifth apart that move to another perfect fifth sound like one horn, and the texture loses a line (see [Voice leading](#)). So each pair of horns that would move in parallel fifths counts as 24 semitones of movement, more than any other way to the next chord, and the horns pick another voicing.

In the code

What the horns play is long notes, or short chords in the rhythm of one of the ways the piano comps:

```
pub enum HornPart {
    /// Long notes: each chord held for as long as it lasts.
    Pads,
    /// Short chords in the rhythm of a way of comping.
    Hits(Comping),
}
```

Comping already had every rhythm the horns need (Charleston, Tresillo, Funk, Stabs, Dembow, Offbeat, Aksak for the afterbeats and Accents), so the horns reuse them.

HornPart::of reads the style: two lists of style names first, then the meter, then the genre:

```

if style.rhythm.is_none() || HELD.contains(&style.name) {
    return HornPart::Pads;
}
if let Some(&(_, comping)) = BY_STYLE.iter().find(|(name, _)| *name ==
style.name) {
    return HornPart::Hits(comping);
}

```

`iter().find` returns an `Option<&(&str, Comping)>`, and the pattern `Some(&(_, comping))` takes the pair apart in one go: `&` for the reference, `_` for the name it does not need. `close` builds one voicing from the top down, starting on the lead and taking the nearest note below that is a chord note no horn plays yet:

```

fn close(lead: u8, classes: &[u8]) -> Vec<u8> {
    let mut notes = vec![lead];
    let mut above = lead;
    while notes.len() < classes.len() {
        let Some(below) = (1..12)
            .filter_map(|down| above.checked_sub(down))
            .find(|note| {
                classes.contains(&(note % 12)) && notes.iter().all(|other|
other % 12 != note % 12)
            })
        else {
            break;
        };
        notes.push(below);
        above = below;
    }
    notes.reverse();
    notes
}

```

`checked_sub` gives `None` instead of wrapping around below note 0, and `filter_map` drops those. `choose` tries every lead from F4 to F5 that is a chord note, builds the close voicing under it, and keeps the cheapest:

```

fn choose(classes: &[u8], last: Option<&[u8]>) -> Vec<u8> {
    (LEAD_LOW..=LEAD_HIGH)
        .filter(|lead| classes.contains(&(lead % 12)))
        .map(|lead| close(lead, classes))
        .min_by_key(|voicing| cost(voicing, last))
        .unwrap_or_default()
}

```

`cost` adds up how far each horn moves, matching the horns from the top down, plus 24 for each pair in parallel fifths. `min_by_key` returns the first of the cheapest, and `unwrap_or_default` gives an empty voicing, silence, for a harmony with no notes. At

most thirteen leads are tried, each with four horns at most, so trying them all is fast; there is no need for anything cleverer.

Following the chords you hold

An **arranger keyboard**, the home keyboard with a “style” button, plays drums, chords and a bass line, and the left hand only tells it *which chord*. Hold C and the band plays in C; move to A minor and the same groove carries on in A minor. **MIDI >**

Accompaniment follows chords does that with any song: play it, hold a chord on a MIDI keyboard (or on the letter keys, with **Typing keyboard to piano**), and from the next note they play, the chords and the bass line move into it. The status bar says which chord the accompaniment is in.

Audio: Default Audio Device - 48000 Hz - 2 channels

 A up to C: a minor third - accompaniment in Am

Like arranger keyboards, it does not need the whole chord:

Keys held	Chord
one key, such as G	its major chord: G
two keys a minor third apart, such as D and F	the minor chord on the lower one: Dm
two keys a major third or a fifth apart	the major chord on the lower one
two keys a minor seventh apart, such as G and F	the seventh chord: G7
three keys or more	the chord they make

The last chord stays when you let go, as the “memory” button of an arranger keyboard does. Only a key going *down* asks for a new chord, so letting go of the keys one by one does not count as playing the keys left.

Before it can move a note, the app needs to know which chord the note was written for. It hears each pattern once, with the second ear of [Hearing harmony](#), and keeps what it heard until the project changes. Then every note of a stretch heard as, say, C major is moved into the chord held, say A minor. Arranger keyboards have a table for this, written by whoever made the style; Ferrous Lineup works it out from what each note *does* in its chord, its role:

```
enum Role {
    Root,
    /// The third, or what stands for it in a "sus" chord.
    Third,
    Fifth,
    /// The seventh, or the sixth of a sixth chord.
    Seventh,
    /// A ninth, eleventh or thirteenth.
    Tension,
}
```

Block chords, three notes or more starting together, are voiced again for the new chord as close as possible to where they were, so the hand hardly moves. Single notes, as in arpeggios and bass lines, keep their role: the root of C goes to the root of A minor, the third (E) to the third (C), the fifth (G) to the fifth (E); a seventh goes to the new seventh, or to the root if the new chord has none. Notes outside the chord, such as the passing notes of a walking bass, move as far as the root does. Each moved note lands in the octave nearest to where it was, which keeps a bass line low and an arpeggio the same shape:

```
fn nearest(pitch_class: u8, near: Note) -> Note {
    let near = i32::from(near.midi());
    let up = (i32::from(pitch_class) - near).rem_euclid(12);
    // Up to a tritone up, or else down.
    let midi = if up <= 6 { near + up } else { near + up - 12 };
    Note::new(midi.clamp(0, 127) as u8)
}
```

`rem_euclid` is the remainder that is never negative: `-3 % 12` is `-3` in Rust, but `(-3i32).rem_euclid(12)` is `9`, which is what “how far up to the next A” means.

The song itself never changes: undo, saving and the WAV export see the notes as written. The moved notes go to the audio thread only, with the command every edit already uses to replace one channel’s notes in one pattern (see [Sending commands with a ring buffer](#)):

```
self.send(Command::SetNotes {
    pattern: moved.pattern,
    channel: moved.channel,
    notes: swing::swung(&moved.notes, swing, hits),
});
```

The audio thread swaps the list between two blocks of sound, so the next note it starts is in the new chord; notes already sounding ring out as they were. When playback stops or the option is turned off, the project’s own notes are sent back. “Only

a key going down” is one flag, set on each Note On and read with `std::mem::take`, which returns it and leaves `false` behind:

```
if !std::mem::take(&mut self.live.follow.key_went_down) {  
    return;  
}
```

In the file

A project with a band saves its style by name, where it ends, its channels, the fingerprints, and what it muted. A new song from **File > New** with C, Am, F and G7 on its sheet saves this:

```
band: Some((  
    style: "Pop beat",  
    end: 64,  
    piano: Some(0),  
    bass: Some(1),  
    guitar: Some(2),  
    strings: Some(3),  
    written: (Some(14171519900600248065), Some(11668178586522433030),  
Some(3477506964485452990)),  
    written_too: (Some(13661597678858777105),  
Some(14695981039346656037), None),  
    heard: Some(6239919094066095261),  
    muted: (),  
    since: 0,  
)),
```

and each band pattern carries `player: Some(Piano)` and so on. The fingerprints are split oddly: `written` holds three, `written_too` three more. The band in memory has one array of eight, but the file has to stay readable by older versions where it can. A band of a piano, a bass and drums was saved as `written: (... , ... , ...)` before there were other players, so those three stay where they were, and the others go in fields of their own that are left out when empty:

```
/// The fingerprints of the piano, the bass and the drums.  
pub written: [Option<u64>; 3],  
/// The fingerprints of the guitar, the strings and the organ, left  
/// out when there are none, so a band of three reads as before.  
#[serde(default, skip_serializing_if = "none_written")]  
pub written_too: [Option<u64>; 3],  
/// The fingerprint of the soloist's part.  
#[serde(default, skip_serializing_if = "Option::is_none")]  
pub written_solo: Option<u64>,
```

`skip_serializing_if` leaves a field out of the file when the function named says so, and `default` fills it in when a file does not have it. Turning the array into the file's fields is one pattern:

```
let [piano, bass, drums, guitar, strings, organ, soloist, horns] =  
band.written;
```

An array pattern takes an array apart into names, one per element, and the compiler checks that there are exactly eight. So a band of three is written exactly as before, and its song keeps the format it needed; a band needs format 13, a bigger band 17, a soloist 19, a horn section 28 and another style's drums 29, saved as `drums:` `Some(Of("Bluegrass"))` or `drums: Some(Song)` (see [The version number](#)), and an older version refuses those with a message instead of silently dropping a player. A file whose band plays a style this version does not know, or on a channel that does not exist, is refused with the reason rather than opened wrong.

Testing it

`band.rs` has the widest tests of the band:
`a_new_song_is_an_empty_sheet_with_a_band`, `the_band_plays_what_is_typed`,
`the_band_follows_every_change_of_the_sheet`,
`an_edited_part_stays_until_it_is_played_again`,
`a_frozen_part_stays_frozen_in_the_file`,
`sets_micromodes_and_rows_are_played_by_their_notes`,
`no_chord_silences_every_style`, `a_style_in_three_puts_the_song_in_three`,
`a_walking_bass_leads_into_each_chord_by_a_semitone` (in Swing and in a jazz waltz), `a_band_in_a_demo_mutes_the_songs_own_accompaniment`,
`deleting_a_players_channel_freezes_only_that_part`,
`long_songs_are_cut_into_patterns_and_repeats_share_one`, and for the style parts
`instruments_can_change_without_changing_the_performance`,
`the_progression_is_independent_of_the_rhythm_and_sounds` and
`separate_style_components_survive_saving_and_undo`.
`every_rhythm_is_a_band_style_and_names_are_unique` reads the whole list. In
`comping.rs`, `a_walking_line_leads_into_the_next_chord` checks C2, E2, G2, F#2
over C7 going to F, `a_walking_line_goes_the_way_of_the_next_chord` the four lines
of C, Am, F, G7 and that, from every chord to every root, the last step is a half step and
no step is wider than a sixth, and
`plain_accents_follow_the_grouping_without_a_styled_bass_line` the beats of an
odd meter.

The guitar's tests check the music: `open_chords_are_the_ones_in_chord_books`,
`other_keys_take_barre_chords`, `a_slash_chord_has_its_bass_lowest`,
`every_chord_in_every_key_has_a_shape_with_its_needed_notes`,
`power_chords_and_shells`,
`a_down_stroke_goes_up_the_strings_and_an_up_stroke_comes_down`,
`a_chop_strikes_the_top_four_strings_on_two_and_four` and
`the_boogie_alternates_fifth_and_sixth`. The horns':
`the_horns_move_as_little_as_they_can` (C, A $\flat$, G7, C as above),
`the_horns_never_move_in_parallel_fifths_when_they_can_help_it`,
`a_big_chord_leaves_out_its_fifth_and_root` and
`every_style_has_horns_that_fit_it`; and in `band.rs`,
`horns_come_to_any_style_when_asked` brings them to six styles and checks that
every horn note is a note of its chord, every voicing is in close position and the lead is
between F4 and F5.

Following a held chord is checked by `keys_ask_for_chords`, `notes_keep_their_role`,
`a_seventh_goes_to_the_root_of_a_triad`,
`every_style_follows_any_chord_in_its_chords` (for every style, the chord channel
plays only notes of the chord held) and `every_demo_song_follows_a_chord`. And the
[playback trace](#) of a band song lists every note each player plays, bar by bar, which is
how a change to the band is heard without listening.

Exercises on the band are in [Exercises: the band](#).

An orchestra for music that changes meter

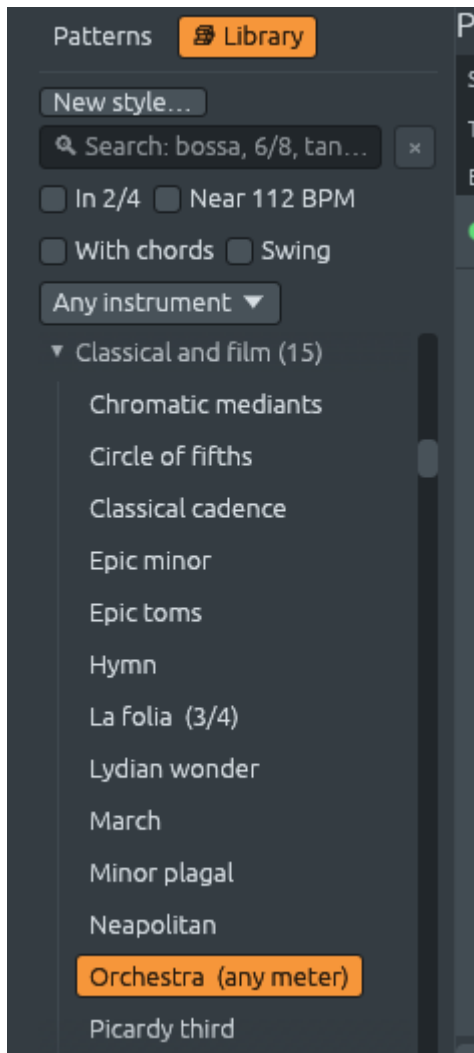
Files: `crates/lineup-core/src/library/rhythm.rs`, `crates/lineup-core/src/library/style.rs`, `crates/lineup-core/src/library/apply.rs`, `crates/lineup-core/src/band.rs`, `crates/lineup-core/src/band/following.rs`, `crates/lineup-app/src/app/library.rs` and `crates/lineup-app/src/library_browser.rs`

Every style in the Library is written in one time signature, and applying it puts the song in that time (see [Putting a song in another time signature](#)). That is right for a pop song, and wrong for The Rite of Spring. The **Ritual of Abduction** changes meter 32 times in 67 bars: a style in 4/4 would turn all of it into 4/4, and a band that mutes the song's own chords and bass would mute Stravinsky.

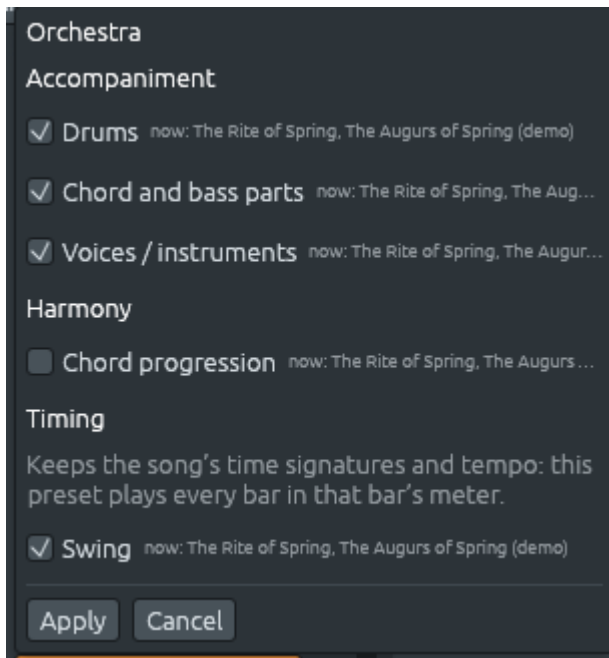
The **Orchestra** style is made for such music. It plays every bar in that bar's meter, keeps the song's time signatures and tempo, and plays along with the score instead of muting it. It is the matching style of eleven demos made from orchestral scores (see [From a MIDI performance or a score to a demo](#)): Promenade, the *Pathétique* waltz in five, Boléro, La Mer, the Augurs of Spring, the Ritual of Abduction and five scenes from Petrushka.

Trying it

1. Open **File > Demos**, type `Augurs` in the search box and press Enter: **The Rite of Spring, The Augurs of Spring** opens and plays. Stop it.
2. Look at the **Library** tab. **Orchestra (any meter)** is lit in the **Classical and film** group, the way a matching style always is. "Any meter" says the style is in no time signature of its own.



3. Click it. Under **Accompaniment**, **Drums**, **Chord and bass parts** and **Voices / instruments** are ticked; under **Harmony**, **Chord progression** is not. Under **Timing** there are no Time signature and Tempo boxes, only a line that says the preset keeps the song's time signatures and tempo.



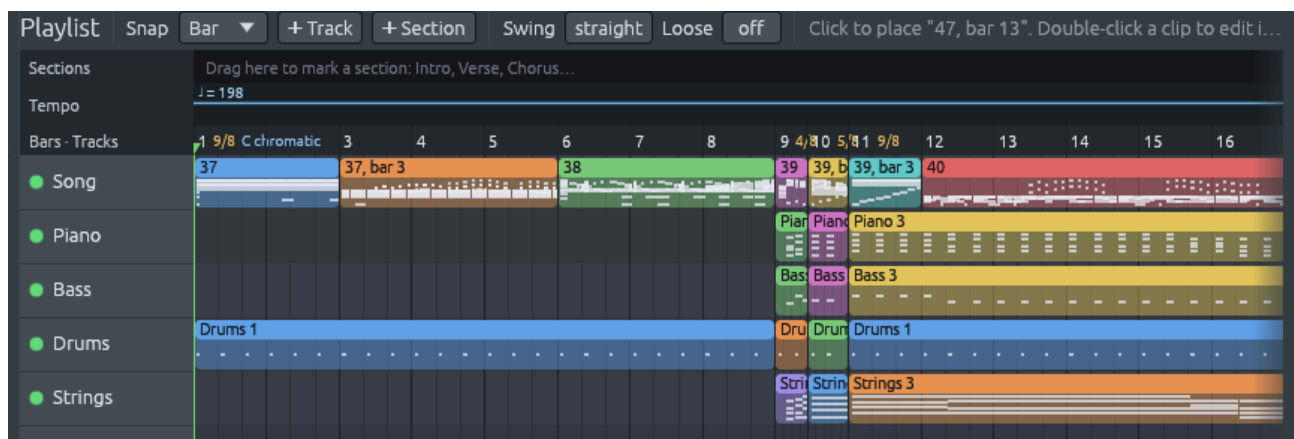
4. Click **Apply**. Four tracks appear under the song: **Piano, Bass, Drums** and **Strings**. The Song track stays on, the Time box still says 2/4 and the tempo still says 112.

5. Press Play. The strings hold each chord under Stravinsky's stamping strings. The horns, the timpani and the bass drum stay silent until bar 3, then strike *with* the score's accents: off the beat twice in bar 3, off the beat in bar 5, on the beat in bars 6 and 7, off it again in bar 8. In the quiet bars 9 to 12 they rest, and the strings play softly.



The tracks keep the band's player names, but in this style the Piano track plays French horns and the Bass track timpani; the Mixer shows the instruments.

Now do the same with **The Rite of Spring, Ritual of Abduction**, which starts in 9/8 at Stravinsky's dotted quarter = 132 (the tempo box shows 198, the same speed counted in quarter notes) and goes through 4/8, 5/8, 12/8, 6/8, 7/8, 3/4, 2/8 and more. The tracks are cut into short clips, one for each run of bars in one meter, and each clip plays its bars the way that meter is felt.



The orchestra plays the chords the app recognised in the score, the ones on the chord sheet (**F5**). Where no chord was recognised, only the bass drum plays. If you tick **Chord progression** in the popup, it plays its own chords instead, I and $\flat$ VII of the Mixolydian mode, the way a style's progression always replaces the song's chords.

Voices alone

Untick **Drums** and **Chord and bass parts** before **Apply**, so that only **Voices / instruments** is ticked, and the same four tracks come in. Tick only the drums, or only the chord and bass parts, and they come in as well: an orchestra that follows the song always brings its own players and its own patterns, on tracks of its own, and the score's channels keep their instruments and their notes.

Other styles treat a single box differently (see [Which parts to take](#)). Picked with its chord and bass parts and its voices but not its drums, a style brings its band in on tracks of its own, and the song's own drums play on. Picked with its voices alone, it brings no band: its sounds go to the song's own players. Picked with its drums or its chord and bass parts alone, it opens in a trial pattern so you can hear it first.

The music

Felt beats

A bar of 9/8 has nine eighth notes, but nobody counts nine. The eighths come in three groups of three, and you feel three beats, each a dotted quarter long. A bar of 5/8 has five eighths in two beats, a short one of two and a long one of three. A bar of 2/4 at a fast tempo goes by as a single beat. These are the *felt* beats, and

`TimeSignature::felt_groups` gives them for any time signature (see [Beats of different lengths](#)).

A conductor beats the felt beats, not every eighth: in the Ritual, three beats for a bar of 9/8, two for 5/8, one for 2/8. In music whose meter changes all the time, that is what keeps an orchestra together, and it is what the Orchestra style plays:

Meter	Felt as	The horns and timpani strike on eighth (or quarter)
2/8, 3/8, 2/4, 3/4	one beat	1
4/8	2 + 2	1 and 3
5/8	2 + 3	1 and 3
6/8	3 + 3	1 and 4
7/8	2 + 2 + 3	1, 3 and 5
9/8	3 + 3 + 3	1, 4 and 7
12/8	3 + 3 + 3 + 3	1, 4, 7 and 10
4/4	2 + 2	quarters 1 and 3

The horns strike the chord on each of those beats and the timpani play its root; when a chord changes between two beats, they play the new one where it starts. The style's own drum line hits the first beat of the bar hard and the others softly, but along with a score the bass drum keeps only the first beat, and only where the music is loud: a bass drum on every beat of a quiet bar would cover the score.

Playing along with the score

A band plays the same groove bar after bar. An orchestra that joins a symphony cannot: the music grows loud and soft, stops, and puts its accents where the composer wants them. The Augurs of Spring is the famous case. The strings stamp

the same chord on every eighth note, all equally hard, and eight horns throw accents onto some of those eighths at places nobody can predict: the second and fourth eighths of one bar, the second of another, the first of the next two. The shock of the piece is that the accents never settle into a pattern. A bass drum on every first beat would turn them into one.

So the Orchestra listens to the score before it plays, and changes what its players would play:

- **Dynamics.** It plays as loud as the score, bar by bar: softer in a *piano* bar, harder in a *forte* one. Where the score is silent, it is silent.
- **Accents.** Where the score accents a chord, the horns strike it, the timpani strike its bass and the bass drum strikes with them, as hard as the score.
- **Passages of accents.** Where accents come close together, in at least two bars out of nine (a bar and the four either side of it), the horns, the timpani and the bass drum play *only* the accents, so the irregular pattern stays the composer's.
- Everywhere else they keep to the felt beats, and the strings hold the chords.

What counts as an accent? A chord that passes both of the first two tests below, or a note that passes the third, the way you would hear them:

1. The whole orchestra plays it much louder than the rest of its bar, *and* than the same place in the bars around it: nearly twice as loud. A tutti chord after a thin texture.
2. An instrument in it plays a short note, between rests: nothing an eighth before it, nothing an eighth after it, and silence after the note ends. The horns in the *Augurs* do that. A melody of quarter notes does not (its notes touch), and neither does a chord held for four bars, which only comes in. That is why *Carol of the Bells*, whose choir holds a chord under the tune, has no accents, and its orchestra plays one stroke a bar from the start.
3. The score marks it itself, a *sforzando* or a strong accent: very hard (velocity 120 or more) and a fifth harder than the middle of its part in that bar.

A bar where most eighths would count as accents has none: the music just got louder.

Why horns and timpani

The style needs a part that marks the harmony on the beat and a part that marks the bass, the jobs a piano and a bass guitar do in a band. In an orchestra those jobs have long gone to the horns and the timpani: in a symphony by Haydn or Beethoven, the loud chords that close a phrase are the horns and trumpets playing the harmony while the timpani strike the tonic or the dominant underneath. The strings hold each chord for as long as it lasts, the way a band's string pad does (see [A bigger band](#)).

The orchestra plays *along* with the score. In a pop demo the band mutes the song's own chords and bass, because two pianos playing the same chords in two different voicings sound like a mistake. In the Rite, the song's parts *are* the piece: muting them would leave only the band. So the Orchestra comes in on top, and every channel of the score keeps its own mute.

How it works

A rhythm that follows the meter

A rhythm in the library is written in one time signature: its drum lines are bars of steps, `x... .. o... ..` for a bar of 4/4, where `x` is a hard hit, `o` a soft one and `.` silence. The band lays a line out bar after bar, and in a song with time markers it writes each run of bars in one meter as a pattern of its own.

The Orchestra rhythm has one line, that 4/4 bar for the kick, and a flag on `Rhythm`:

```
pub struct Rhythm {  
    // ...  
    pub follows_the_meter: bool,  
}
```

The rhythms are built by a `const fn rhythm(...)`, and a builder method sets the flag, the same way `swung` sets the swing:

```
const fn following_the_meter(self) -> Rhythm {  
    Rhythm {  
        follows_the_meter: true,  
        ..self  
    }  
}
```

`..self` is *struct update syntax*: every field not named takes its value from `self`. A `const fn` can do it because it only moves values, so the whole table of rhythms is still built at compile time.

Drum notes for a bar come from `Rhythm::notes_in(drum, time, length)`. For an ordinary rhythm it lays the line out bar by bar, as `Rhythm::notes` does. For one that follows the meter it first writes the line again for `time`:

```

fn in_meter(line: &str, own: TimeSignature, time: TimeSignature) -> String
{
    let steps: Vec<char> = line
        .split('|')
        .next()
        .unwrap_or_default()
        .chars()
        .filter(|character| !character.is_whitespace())
        .collect();
    let first = steps.first().copied().unwrap_or('.');
    let other = felt_starts(own)
        .get(1)
        .and_then(|&start| steps.get(start).copied())
        .unwrap_or('.');
    let mut bar = vec!['.', time.steps_per_bar()];
    for (index, start) in felt_starts(time).into_iter().enumerate() {
        if let Some(step) = bar.get_mut(start) {
            *step = if index == 0 { first } else { other };
        }
    }
    bar.into_iter().collect()
}

```

It reads the line's first bar (a line can hold several, split by `|`), keeps the hit on its first step, `first`, and the hit on the start of its second felt beat, `other`, then lays them out again on the felt beats of `time`. `felt_starts(time)` turns the felt groups into steps: a group of three eighths is three times `time.unit_steps()`, the steps one eighth lasts. For 9/8 that gives steps 0, 6 and 12, and the 4/4 bar's `x` and `o` land there:

```
X.....o.....o.....
```

A rhythm that follows the meter has no time signature of its own, so

`Rhythm::for_time`, which picks a rhythm for a song's new time signature, never picks it.

A style that follows the song

A style knows whether it follows the song by asking its rhythm:

```

pub fn follows_the_song(&self) -> bool {
    super::Rhythm::named(self.rhythm).is_some_and(|rhythm|
    rhythm.follows_the_meter)
}

```

`Option::is_some_and` is `true` only when there is a value and the closure says yes for it, so a style whose rhythm is missing does not follow anything.

Applying a style can put the song in the style's time and tempo, when those boxes are ticked. The parts to apply, `StyleParts`, have a step that drops both when the style follows the song:

```
pub fn following(mut self, follows: bool) -> Self {
    if follows {
        self.time_signature = false;
        self.tempo = false;
    }
    self
}
```

The Library's **Apply** goes through it, and `Project::set_band_style` asks `follows_the_song` before it puts the song in the style's time, so neither re-meters the Rite. The chord sheet's **Style** menu for a section, which only offers styles in the bar's own time signature, offers the Orchestra in any meter. In the app the same flag swaps the Timing boxes for the line you saw in the popup, and the Library adds "(any meter)" to the style's name.

Joining the song

The Library's **Apply** ends up in `band_plays_parts` in `crates/lineup-app/src/app/library.rs`. It decides first whether a band comes in at all:


```

let follows = match action {
  LibraryAction::Style(style) | LibraryAction::StyleInSong(style) => {
    style.follows_the_song()
  }
  LibraryAction::Rhythm(rhythm) | LibraryAction::RhythmInSong(rhythm) =>
{
  rhythm.follows_the_meter
}
  _ => false,
};
let has_music = !self.project.playlist().chord_markers().is_empty()
|| self.project.playlist().clips().iter().any(|clip| {
  self.project
    .pattern(clip.pattern)
    .is_some_and(|pattern| !pattern.is_empty())
});
// A beat made up in the library has no chord and bass parts of its
// own: it brings no band, only its drums.
let made_up = match action {
  LibraryAction::Rhythm(rhythm) | LibraryAction::RhythmInSong(rhythm) =>
{
  BandStyle::named(rhythm.name).is_none()
}
  _ => false,
};
let joins = if follows {
  parts.rhythm || parts.matching_style || parts.instruments
} else {
  parts.instruments && parts.matching_style && has_music && !made_up
};
if self.project.band().is_none() && !joins {
  return false;
}

```

`match` with `|` between two patterns runs the same arm for both, and `_ => false` covers every other kind of Library item. `has_music` is true when the song has chords on the chord sheet, or any notes whose chords the band can hear; `Iterator::any` stops at the first clip whose pattern has a note. A band plays the chord and bass parts of a style, so it comes in only when they are taken, with the voices it plays them on; the drums alone are a matter for the song's own drum channels. When `band_plays_parts` returns `false`, the caller goes on the usual way: it opens the style in a trial pattern, or, while the song plays, applies it to the song's own patterns.

The same rule is in `Project::apply_band_parts` in `band.rs`, which builds the band, with one more line: a band that came in without the style's drums leaves them to the song.

```
if !parts.rhythm && !follows {
    band.drums = BandDrums::Song;
}
```

Listening to the score

`crates/lineup-core/src/band/following.rs` is a module of the band. `band.rs` declares it with `mod following;`, and Rust finds it in `band/following.rs`, a folder named after the parent file. Its type is `pub(super) :` visible to `band.rs` and nothing else.

`Following::hear(project, bars)` collects what the song plays. It walks the clips that are heard, skips the band's own patterns, and records every note's start in a `BTreeMap<usize, Vec<(usize, u8, usize)>> : step to a list of (channel, velocity, length)`. A `BTreeMap` keeps its keys in order, so `struck.range(bar.start..bar.end())` gives the notes of one bar, in time order, without looking at the others.

It returns an `Option<Following> : None` when the song plays no note at all, a chord sheet with nothing else. There is no score to follow then, and the orchestra plays as its style wrote. The same goes for the bars before the score's first note and after its last one ends:

```
let first = *struck.keys().next()?;
let last = struck
    .iter()
    .flat_map(|(&at, notes)| notes.iter().map(move |&(_, _, length)| at +
length.max(1)))
    .max()?;
```

The `?` after `next()` and `max()` returns `None` from `hear` when there is nothing, the same way it passes an error up from a function that returns a `Result`. `flat_map` turns each step's list of notes into the steps where they end, one long iterator, and `max` takes the last. The `move` before the inner closure copies `at` into it: the inner iterator is used after the outer closure has returned, when its own `at` is gone.

How loud a bar is: the loudest note at each step it plays, and the middle one of those, divided by 100 (mezzo-forte in the demos).

```
fn middle(mut values: Vec<u32>) -> u32 {
    if values.is_empty() {
        return 0;
    }
    values.sort_unstable();
    values[(values.len() - 1) / 2]
}
```

The middle value (the *median*) rather than the average, because one loud accent should not make the whole bar loud.

The thresholds are written as fractions, so all the arithmetic stays in whole numbers:

```
const TUTTI: (u32, u32) = (9, 5);
// ...
let (times, by) = TUTTI;
let stands_out =
    strength(at) * by >= in_bar * times && strength(at) * by >= elsewhere
    * times;
```

`strength(at) * 5 >= in_bar * 9` is `strength(at) >= 1.8 * in_bar` without a floating-point number in sight. `elsewhere` is the same place in the two bars before and the two bars after, the quieter of the two sides; the first bar has no bars before it, so it compares with the bars after only:

```
let elsewhere = match (same_place(before.clone()),
    same_place(after.clone())) {
    (Some(before), Some(after)) => before.min(after),
    (one, other) => one.or(other).unwrap_or(0),
};
```

Matching on a pair of `Option`s handles all four cases in two arms: the first takes both values, the second whichever exists, or 0.

To know whether a channel strikes “alone”, `hear` keeps each channel’s starts in order, and finds the note’s place among them with `partition_point`, a binary search: the number of starts before `at`.

```
let place = steps.partition_point(|&step| step < at);
let before = place
    .checked_sub(1)
    .is_none_or(|before| steps[before] + ALONE < at);
let after = steps
    .get(place + 1)
    .is_none_or(|&after| after > at + ALONE.max(length));
```

`checked_sub(1)` is `None` for the first note instead of underflowing below zero, and `Option::is_none_or` is `true` for `None`: no note before the first one is as good as a rest.

`Following::play_along(player, notes, ...)` then takes what a player of the style wrote and changes it: notes in a silent bar are dropped, the horns' and timpani's notes in a passage of accents are dropped, the bass drum keeps only the first beats of loud bars, and every velocity is scaled by the bar's loudness. Then it adds the strikes: at each accent, the horns play the notes of the chord that is held there, the timpani its bass, and the bass drum a hit, as hard as the score's accent.

`write_band`, which writes every player's part, calls it only for a style that follows the song:

```
let mut part = self.band_notes(&playing, player, &score);
if let Some(following) = &following
    && playing.style.follows_the_song()
{
    part = self.played_along(following, &playing, player, part, &score);
}
```

`if let ... && ...` is a *let chain*: the `if` runs only when the pattern matches and the condition after `&&` holds.

Many short patterns

The band writes a pattern for each player and each run of bars in one meter. Two runs can share a pattern when they play the same notes, as the drums of two runs of 9/8 do, but the chords change from run to run. The Ritual of Abduction has 33 runs, and its orchestra writes 92 patterns on top of the score's own 34. A project holds up to 999 patterns (`MAX_PATTERNS` in `project.rs`), and the twelve-tone window reads the same constant when it says a row would need more patterns than a project can hold.

Testing it

The tests in `band.rs`:

- `an_orchestra_plays_every_bar_of_the_rite_of_spring_in_its_own_meter` checks that the Ritual keeps every meter change, its tempo changes and every channel's mute, that the players are a French horn, timpani and a string ensemble, that the bass drum plays only with the score's accents or on the first

beat of a bar, and not in every bar, and that the horns and timpani play on felt beats, chord changes or accents.

- `the_orchestra_strikes_with_the_accents_of_the_augurs_of_spring` copies only the voices and checks that the song's channels and notes stay as they were, that the horns, timpani and bass drum strike at steps 18, 22, 34, 40, 48 and 58 in the first eight bars and nowhere else, and that in bars 9 to 12 they rest and the strings play below velocity 60.
- `a_tune_or_a_held_chord_is_not_an_accent` hears no accents in Carol of the Bells, and checks the horns play the first beat of each of its first eight bars.
- `with_no_score_to_follow_the_orchestra_plays_as_written` applies it to a chord sheet with no notes, and checks every player plays.
- `a_band_brought_without_drums_leaves_them_to_the_song` copies the chord and bass parts and the voices of a style that does not follow the song, Emotional pop, onto the tango El Choclo, without its drums, and checks its band comes in with no drums of its own while the song keeps its sounds, its notes and its drums; taking the drums later brings them in place of the song's.
- `copying_only_the_voices_brings_no_band` hands only the voices of Emotional pop to `apply_band_parts` on El Choclo, and checks it leaves the song as it was: voices alone go to the song's own channels, by the Library's own route, not through a band.
- `picking_the_orchestra_keeps_the_meters_of_the_song` makes the Orchestra the band's style in the Augurs, with `set_band_style`, and then **Pop beat**: the first keeps 2/4, the second puts the song in 4/4.

In `rhythm.rs`, `a_rhythm_that_follows_the_meter_plays_each_bar_in_its_own` checks the hits `notes_in` writes for 2/4, 3/4, 9/8, 5/8, 7/8 and 3+2+2/8, and that `for_time` never picks the Orchestra.

In the app,

`copying_only_the_voices_or_the_patterns_brings_the_band_and_leaves_the_song_alone` in `ux_tests.rs` clicks through the Library popup for the Augurs, the Ritual, Petrushka's Room, Carol of the Bells and a tango, with one box ticked, and checks the band came in and the song's channels and notes did not change.

To hear it without listening, save the Augurs after **Apply** and read its [playback trace](#):

```
ferrous-lineup --trace augurs.lineup --out augurs.tsv
awk -F'\t' '$11 == "Kick" && $3 <= 16 { print $3, $4, $5, $16 }'
augurs.tsv
```

Column 3 is the bar, 4 the beat, 5 how far into the beat in steps (a quarter note has four, so 2 is the eighth after the beat) and 16 the velocity:

```
3 1 2.000 127
3 2 2.000 127
5 1 2.000 127
6 1 0.000 127
7 1 0.000 127
8 1 2.000 127
14 1 2.000 127
14 2 2.000 127
16 1 2.000 127
```

Off the beat twice in bar 3, off it in bar 5, on it in bars 6 and 7, off it in bar 8: the horns' accents as Stravinsky wrote them, with nothing in the quiet bars between 8 and 14.

Song form, sections, variations and locks

Files: `crates/lineup-core/src/band/form.rs`, `crates/lineup-core/src/band/variation.rs`, `crates/lineup-core/src/band/sections.rs`, `crates/lineup-core/src/band/locks.rs`, `crates/lineup-core/src/lock.rs`, `crates/lineup-core/src/band.rs`, `crates/lineup-core/src/arrangement.rs`, `crates/lineup-core/src/swing.rs`, `crates/lineup-app/src/chord_sheet.rs`, `crates/lineup-app/src/chord_sheet/form.rs`, `crates/lineup-app/src/playlist/sections.rs` and `crates/lineup-app/src/channel_rack.rs`

A band does not play a tune once, the same way from the first bar to the last, and stop dead. It plays a few bars to set the key and the groove, then the tune, maybe several times, and then it ends on the home chord. It plays the verse lightly and digs into the chorus. A Latin jazz tune may play its verse as a bossa nova and its bridge in swing. And if you play the piano part yourself, the band plays around you.

This chapter is about those four tools: the **form** (an intro, choruses and an ending), the two **variations** of every style, a **style for each section**, and **locks**, which keep a track out of the styles' way. The first three live on the chord sheet; the locks live in the channel rack.

The form: an intro, choruses and an ending

Chord sheet

Band: Pop beat Progression Key: C major - 4/4 - 110 BPM

Form Intro: 4 bars Choruses 3 Ending

Count-in Metronome Speed 100% Tap You play Piano Bass Drums Loop

Click a bar and type its harmony: chords (Am7 D7, G/B), Roman numerals (ii7 V7), sets (3-11 T2, {0,1,4}), micromodes (Menor 3), row forms (P0, R17) or N.C.

Each / holds the harmony before it one more beat, as in C // G7.

Intro			
1 4/4	2	3	4
C	Am7 vi7	Dm7 ii7	G7 V7

Chorus x3			
5	6	7	8
C	Am vi	F IV	G7 V7

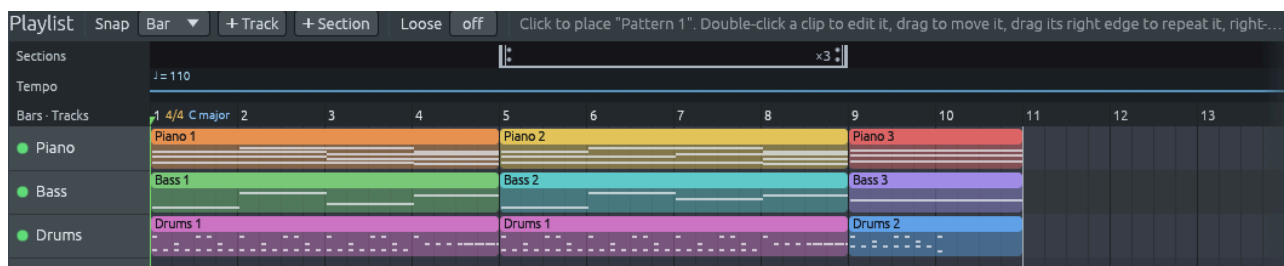
Ending			
9	10	11	12
C	C held		
13	14	15	16

Trying it

Write a few bars of chords on the sheet (**F5**), or pick a progression. Under the sheet's buttons there is a row named **Form**:

- **Intro**: No intro, 2, 4 or 8 bars. The band puts that many bars in front of the song, with chords that lead into its first chord. They are written on the sheet under the heading **Intro**, so you can read them, hear them (hold one down with the mouse) and change them. Picking a length again writes them anew; **No intro** takes them away. The rest of the song, melody and all, moves along to make room.
- **Choruses**: how many times the band plays the tune. The sheet shows it once, under **Chorus x3**, between the repeat signs of printed music: **||**: at the start, **:||** at the end. While it plays, the form row shows which time round it is, "► 2 of 3".
- **Ending**: two bars after the last chorus. The band goes to the key's home chord, plays one bar of it in the style, and then strikes it once and holds it for the whole last bar, with a crash of the cymbal. The two bars are shown in grey under **Ending**, the last one marked "held", with the final double bar line after it.

The playlist shows the repeat too: along the bottom of the section strip, the same repeat signs at the two ends of the chorus, and "x3".



“Chorus” means one time through

In a pop song the chorus is the part that comes back after each verse (in “Let It Be”, “When I find myself in times of trouble” starts a verse and “Let it be, let it be” is the chorus). Jazz musicians use the word differently: a **chorus** is one time through the whole tune, from the top of the chart to its last bar. A soloist “takes two choruses”; the band plays “the head” (the melody) for one chorus, then solos for several, then the head again. The form row uses the word that way: one chorus is every bar on the sheet after the intro. A pop song’s verses and choruses are all inside one chorus here. If your song has sections of its own (Verse ×2, Chorus) on the playlist, they repeat as they always do, inside each time round.

The intro: a turnaround

A **turnaround** is the handful of chords at the end of a tune that “turn it around” and lead back to the top. It is the most natural intro there is: it leads into the first chord, so when the tune starts it sounds like an arrival, not a beginning out of nowhere.

The engine of a turnaround is the **dominant seventh**. In C major, G7 (G B D F) holds two notes that want to move: B, a semitone under C, pulls up to C, and F, a semitone over E, pulls down to E. Played before C, G7 makes C sound like home. And the chord a fifth above G, D minor, leads to G the same way G leads to C: roots that fall by fifths (A → D → G → C) are the strongest way chords can move.

Intro	Into C major	Into A minor
2 bars	Dm7, G7 (ii7 V7)	Bm7b5, E7 (iiø7 V7)
4 bars	C, Am7, Dm7, G7 (I vi7 ii7 V7)	Am, Fmaj7, Bm7b5, E7 (i bVI maj7 iiø7 V7)
8 bars	the four chords, two bars each	the same, two bars each

Into a minor chord the ii is **half-diminished** (B D F A over Am): that is the chord that is really on the second note of A minor. The V is still a dominant seventh, E7, with G sharp: the **leading note** a semitone under A, borrowed from the harmonic minor scale, because a plain E minor would not pull to A nearly as hard.

The numerals are counted from **the chord the intro leads to**, as if it were the tonic, not from the key of the song. Most songs start on their tonic, and then the two are the same. A song in C that starts on F (IV) gets Gm7, C7 for a two-bar intro: the ii and V of F, so the intro still leads straight into the first chord. A blues that starts on C7 gets C7, Am7, Dm7, G7. The chords are spelled from the letter the first chord is written with: into Ebm the four-bar intro is Ebm, Cbmaj7, Fm7b5, Bb7, not the same sounds written from D sharp.

A pitch-class set, a micromode or a row form has no dominant to lead into it, so the intro plays it from the first bar: a **vamp**, the band setting up the sound before the tune comes in. So does a diminished chord, which no turnaround resolves to.

The ending: home, and held

The ending goes to the **tonic**, the chord the whole key leans towards. Which tonic chord? If the tune starts on the tonic, the ending plays that very chord: a blues in C that starts on C7 ends on C7, a ballad that starts on Cmaj7 ends on Cmaj7. Otherwise it plays the key's own triad, C in C major and Am in A minor. The key is the one the last chorus ends in, so a song that modulates ends in its new key. A song in a set or a row has no tonic; it ends where it started, on its first harmony.

The last bar is played differently from all the others: the piano strikes the chord once and holds it, the bass holds the root, and the drums play one kick with the crash cymbal, the library's "Crash ending". The chorus's last bar already gets a drum fill (it leads into the next time round or into the ending), so the ending's first bar is plain groove, and the song does not end with two fills in a row.

The ending is not written on the sheet: its chords are the band's, worked out each time from the key. Type a chord in an ending bar and it becomes a bar of the tune: the chorus grows to take it in, and the ending moves on after it.

Form: three numbers that travel with the band

```
#[derive(Debug, Clone, Copy, PartialEq, Eq, Serialize, Deserialize)]
#[serde(default)]
pub struct Form {
    /// How many steps of intro come before the chorus: 0 for none.
    pub intro: usize,
    /// How many times the chorus plays: 1 to [`MAX_REPEATS`].
    pub choruses: usize,
    /// Whether the band plays an ending after the last chorus.
    pub ending: bool,
}
```

`Form` lives inside the `Band`, because only a band plays one. It is `Copy`: three plain numbers cost nothing to copy, so functions take a `Form`, not a `&Form`. `#[serde(default)]` on the struct means any field missing from a file takes its value from `Form::default()`.

Where are the intro and the chorus? Only the intro's length is stored, in steps. The chorus is worked out when needed, from the end of the intro to the end of the last bar written:

```
pub fn chorus(&self) -> Option<(usize, usize)> {
    let band = self.band.as_ref()?;
    let (start, end) = (band.form.intro, band.end);
    (start < end).then_some((start, end))
}
```

so typing a chord after the last bar makes the chorus longer with nothing to keep in step. The ending is worked out the same way (`Project::ending`): the two bars after the chorus, and the harmony `ending_harmony` picks.

The intro moves the song

Putting bars in front of a song means moving everything else later: clips, sections, markers of time, tempo, key and chords, the loop region. `Project::make_room` does this, the same function that makes room when you copy a section. Taking the intro away needs the opposite, `Project::cut_start`, which also has to keep what the song sounds like where the cut ends: the song then *starts* in the time signature, tempo, key and harmony that were sounding there.

```
let made = self.moving_the_band(|project| {
    project.cut_start(old);
    steps == 0 || project.make_room(0, steps)
});
```

`moving_the_band` runs the change and then tells the band that its parts moved with everything else (see [Moving the whole song is not an edit](#)). Without it, the band would compare its clips with the fingerprints it kept, find them in new places, and think you had edited them: every part would be frozen.

Playing the chorus again: the arrangement

The playlist is the song *as drawn*; the arrangement (see [Sections, repeats and loops](#)) is the song *as played*, with every repeat unrolled. It plays a section marked ×2 twice. The chorus is one more layer around that:

```
let mut played: Vec<_> = once(from, start).collect();
for chorus in 1..=choruses {
    played.extend(
        pieces(project, start, end)
            .into_iter()
            .map(|piece| (piece, Some((chorus, choruses)))),
    );
}
played.extend(once(end, to));
```

The intro's pieces once, the chorus's pieces `choruses` times, then the ending once. Each pass remembers which time round it is in `Pass::chorus`, which the form row and the playlist read for "► 2 of 3".

Because everything that plays reads the arrangement (the audio thread, the WAV and MIDI exports, the metronome's clicks, the tempo of every step), none of them needs to know about choruses. A loop region inside the chorus does not take in all of it, so the chorus is not repeated there: looping bars 3 to 4 plays bars 3 to 4.

The last bar: one closure, two ways of playing

The piano plays every bar with the style's comping, and the last bar of an ending with `Comping::Held` instead; the bass the same with `Bass::Held`. So the band plays its bars in **runs** that leave the held bar out (more on runs under [Playing in runs](#)), and then plays the held bar on its own. The way of playing is handed to the function as a **closure**:

```
fn held_bar(
    held: Option<Bar>,
    harmonies: &[Played],
    play: impl Fn(&[Bar], &[Played]) -> Vec<NoteEvent>,
) -> Vec<NoteEvent> {
    held.map_or_else(Vec::new, |bar| {
        play(&[bar], &cut(harmonies, bar.start, bar.end()))
    })
}
```

`impl Fn(&[Bar], &[Played]) -> Vec<NoteEvent>` means “anything that can be called with bars and harmonies and gives back notes”. The piano passes `|bars, played|` `Comping::Held.play_in_bars(bars, played)`, the bass `Bass::Held`’s `cut` cuts the harmonies to the bar, and also stops a harmony cut short from “leading” anywhere: a walking bass would otherwise walk towards the next chord before it arrives.

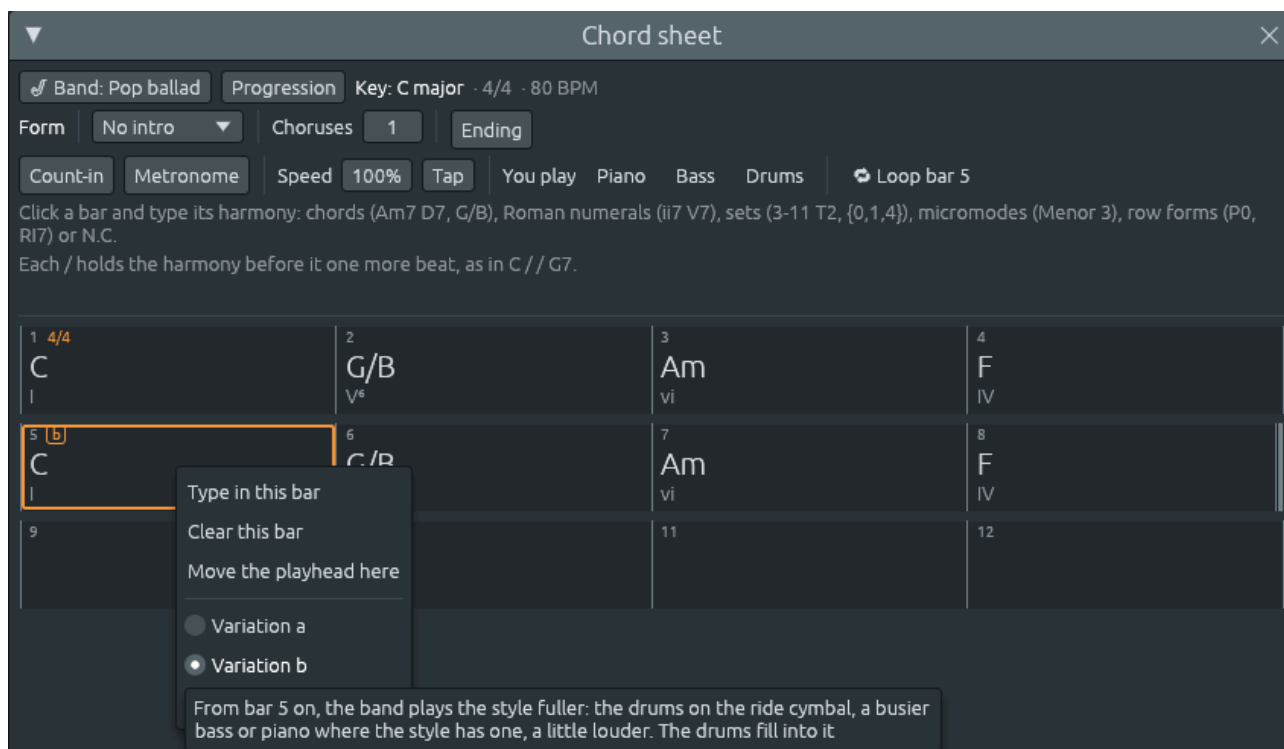
The ending’s harmony is not on the sheet, so the band adds it to the list of markers it plays from, as if it were written there:

```
if let Some(ending) = ending {
    markers.push(ChordMarker::new(ending.start, ending.harmony));
}
```

It is then voiced from the chord before it, like any other, so the piano moves to the last chord smoothly.

Variations a and b: the band digs in

A band does not play every part of a song the same way. The verse is lighter, so the words come through; the chorus is fuller, and the drummer, the bassist and the pianist all lift at once. A jazz group plays the first chorus of a ballad gently and then starts to swing harder. So every style has two variations: **a** is the style as it is, and **b** is the same style played fuller.

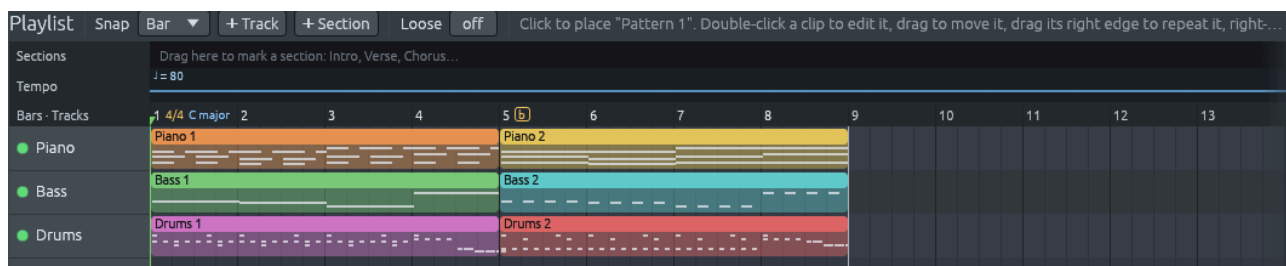


Trying it

Right-click a bar on the chord sheet (**F5**) and pick **Variation a** or **Variation b**. The band plays that variation from that bar up to the next change, or to the end of the song. To change only some bars, select them first: click the first, Shift+click the last, then right-click one of them. The bars after the selection keep what they played before. That is how you play only the bridge of a song fuller, or only the last chorus. Right-clicking a section's name on the sheet selects its bars and opens the same menu.

A boxed letter marks each change, on the sheet after the bar number and on the playlist's bar ruler, like the rehearsal letters printed over the staff in band parts. Picking the variation that is already playing puts no marker; changing a bar back takes the marker away.

The drums announce each change: the bar before it is a **fill**, and the first beat of the new variation gets a **crash** cymbal. This is the same signal real drummers give, and why a listener knows that "something new starts here" even without hearing the words.



What “fuller” means

Variation b does not invent a new style. It opens the same one up, the way musicians do it:

Who	Variation a	Variation b
Drums	time on the closed hi-hat	time on the ride cymbal, unless the style keeps time on the ride already
Drums, in a pop or rock ballad	cross-stick on the snare’s rim	the full snare
Bass, a jazz style played in two, in 4/4	in two : root and fifth, one every two beats	walking : a note on every beat
Bass, blues shuffle	quarter notes	the boogie line, up and down the chord
Bass, pop, rock, soul or folk in 4/4	the root held	the root on every beat
Piano, pop or rock ballad	the chord, then its notes one by one	steady eighth notes
Guitar	chords left to ring, or picked; held power chords	strummed; power chords chugged in eighths
Strings or organ, in some styles	silent	they join in
Everyone		a little louder: each note’s velocity a tenth higher

A few words from that table:

- The **hi-hat** is two cymbals that the drummer closes with a pedal. Closed, it gives a short, dry “tick”. The **ride** is one large cymbal that rings; eighth notes on it

make a shimmering wash that fills more of the sound. Moving from one to the other is the most common way a drummer lifts a chorus.

- A **cross-stick** is the stick laid across the snare drum, its end clicking on the rim: a soft, woody sound. Hitting the snare's head instead is much louder. Ballads are full of cross-stick verses and full-snare choruses.
- A jazz bassist **in two** plays on beats 1 and 3, usually the root and the fifth, which leaves room around a quiet melody. **Walking** fills every beat with a line that leads to the next chord: over Dm7 going to G7, the Jazz ballad's bass plays D, F, A and F#, the F# a semitone under the G it is heading for. Going from "two" to "four" is how a jazz rhythm section turns up the heat without playing louder.

Where a style has no well-known busier way to play (a tango, a bossa nova, a chorale), variation b keeps its patterns and only changes the drums and the loudness. The rules are deliberately few, so that b always sounds like the same band, not like a different song. The guitar's and the pads' parts in b are described with the bigger band (see [Enums that carry data, and a function as a parameter](#)).

Markers that only mark changes

```
pub struct VariationMarker {  
    pub start: usize,  
    pub variation: Variation,  
}
```

The markers live in the `Band`, beside its `Form`, because only a band plays variations. A marker that changes nothing (variation a at the start of the song, or b right after another b) would only clutter the sheet, so after every change `tidy` keeps only the real ones:

```
pub(crate) fn tidy(markers: &mut Vec<VariationMarker>) {  
    markers.sort_by_key(|marker| marker.start);  
    markers.dedup_by(|later, earlier| {  
        later.start == earlier.start && {  
            *earlier = *later;  
            true  
        }  
    });  
    let mut playing = Variation::A;  
    markers.retain(|marker| {  
        let changes = marker.variation != playing;  
        playing = marker.variation;  
        changes  
    });  
}
```


First in order of their start; then, of two markers on the same step, the later one wins (`dedup_by` hands the closure the later element and the one kept before it, and the closure copies the later one over); then only the changes stay. The closure passed to `retain` changes `playing`, a variable outside it. A closure that changes what it captures is an `FnMut`: it can be called many times, but not from two places at once, because each call needs the variable to itself. `retain` asks for exactly that.

Which variation plays at a step is the last marker at or before it:

```
markers
  .iter()
  .rev()
  .find(|marker| marker.start <= step)
  .map_or(Variation::A, |marker| marker.variation)
```

`rev()` walks the list backwards, so `find` stops at the first marker that is not after the step.

Playing in runs

The bars are grouped into **runs**, stretches in one variation, and each run is played with that variation's way of playing:

```
fn played_in_runs(
  runs: &[Run],
  harmonies: &[Played],
  play: impl Fn(Variation, &[Bar], &[Played]) -> Vec<NoteEvent>,
) -> Vec<NoteEvent> {
```

The piano's part is the two calls together, the runs and then the held bar of an ending:

```
Player::Piano => {
  let mut notes =
    played_in_runs(&score.runs, &score.harmonies, |variation, bars,
    played| {
      band.comping_in(variation).play_in_bars(bars, played)
    });
  notes.extend(held_bar(score.held, &score.harmonies, |bars, played| {
    Comping::Held.play_in_bars(bars, played)
  }));
});
```

Each run gets only its own harmonies, cut at its edges with `cut`. A song with no markers is one run.

`comping_in` and `bass_in` are small `match` es on the variation, the style's genre and its way of playing:

```
match (variation, self.genre, self.bass) {
  (Variation::B, Genre::Jazz, Bass::RootFifth) if common =>
  Bass::Walking,
  (Variation::B, Genre::Blues, Bass::Quarters) if common =>
  Bass::Boogie,
  // ...
  (_, _, bass) => bass,
}
```

Matching on a tuple lets one line name a whole situation, and the last arm keeps everything else as it is.

Drums on other drums

The drums keep the rhythm's patterns and change which drum plays them:

`drum_in(Drum::HiHat, Variation::B)` is `Drum::Ride` when the rhythm has no ride of its own. The band writes each pattern into the channel of the drum that should sound, so the hi-hat's eighth notes land on the ride's channel. A channel the song does not have yet (a ride, a crash) is added the first time a variation needs it, and never otherwise: a song without markers keeps exactly the channels it has.

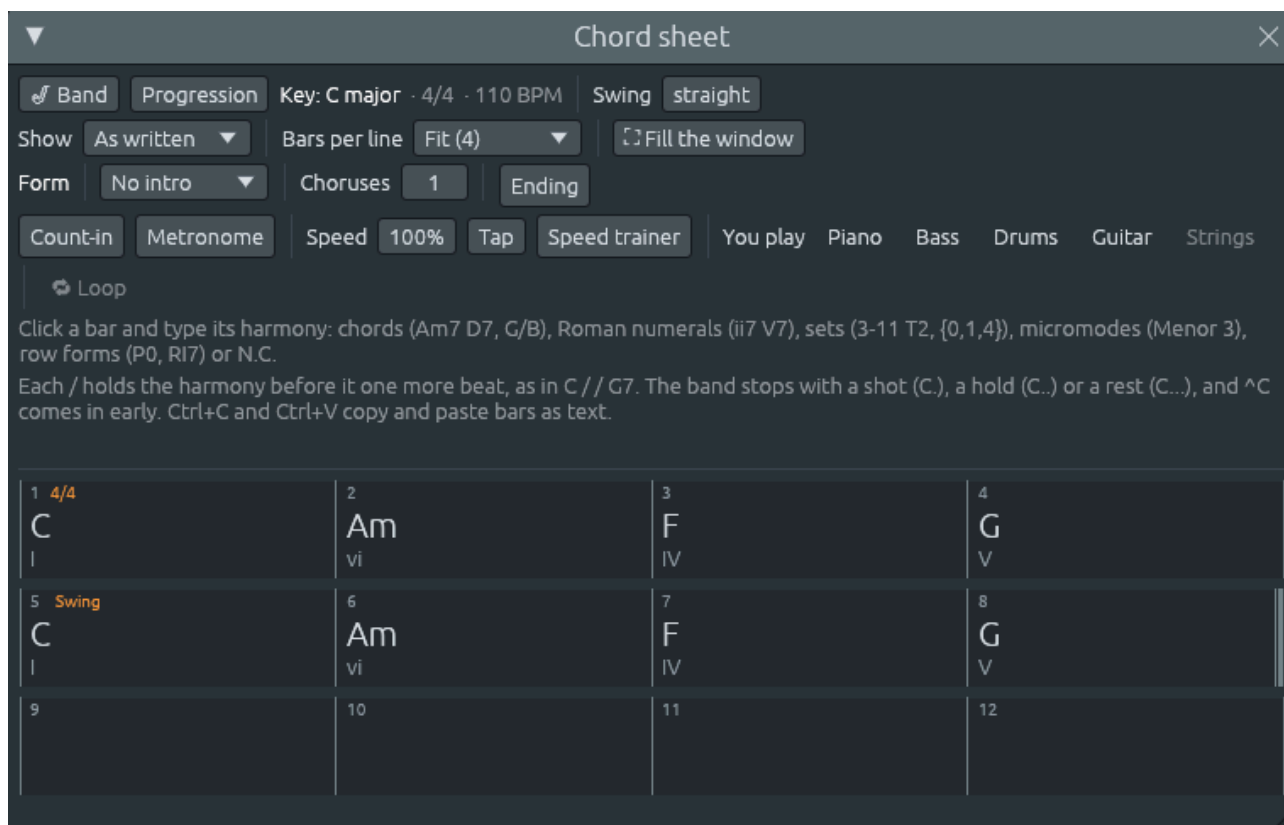
Each change of variation also starts a new **chunk**, the stretch one pattern covers, so the playlist shows the a and b parts as separate clips you can see, mute or edit.

Markers move with the song

Everything that moves the song moves the markers too: an intro put in front (`make_room`), an intro taken away (`cut_start`), a section copied (`copy_section`, which copies the variations inside it), a new time signature (`remeter`), and **End the song here**, which drops the markers after the end.

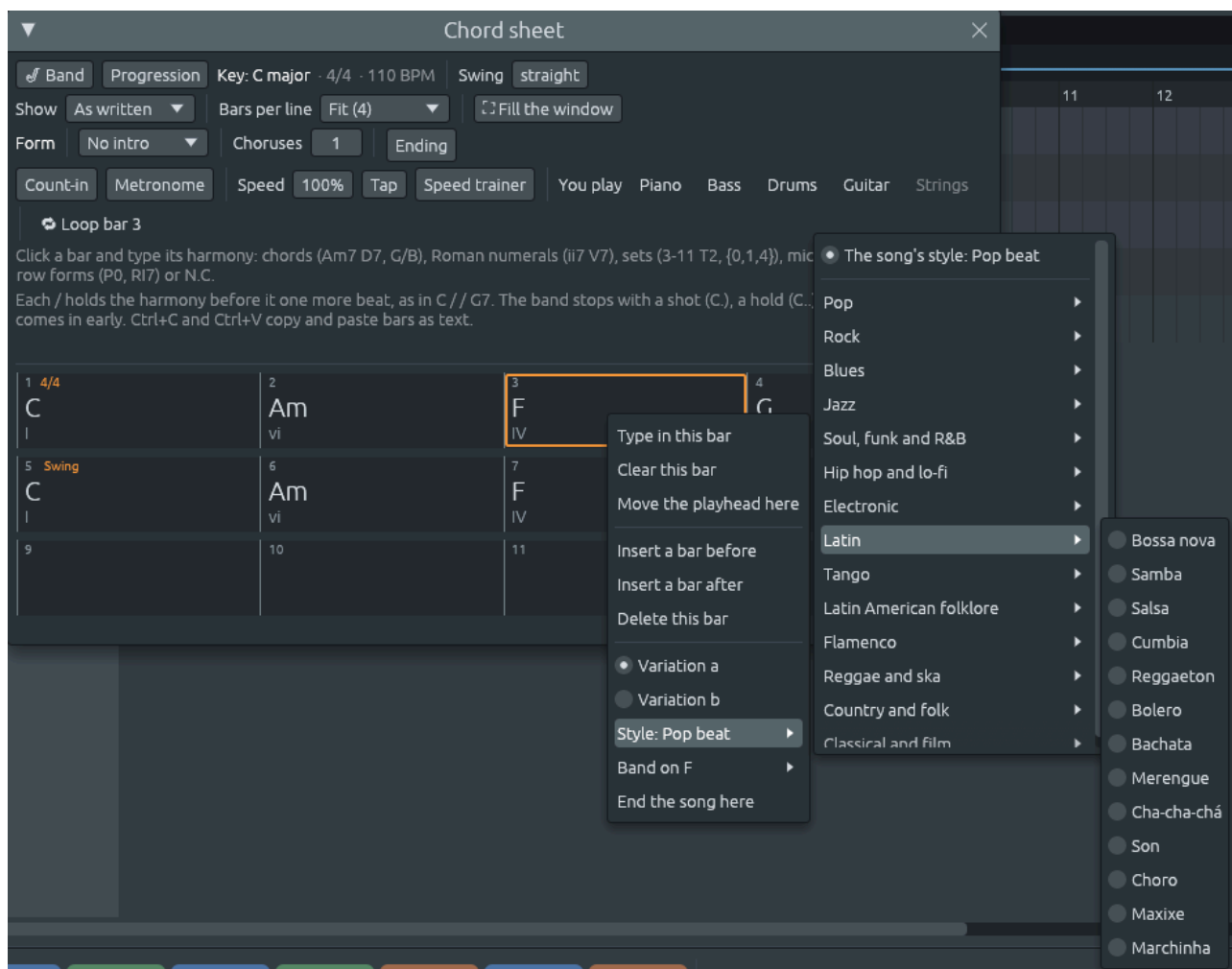
A style for each section

Variations make a part of a song fuller, but in the same style. Many songs change their feel partway through: a Latin jazz tune plays its verse as a bossa nova and its bridge in swing; a pop song goes to half time for the bridge; an arrangement starts as a ballad and ends as a rock song. So each part of a song can have a style of its own.



Trying it

1. Open the chord sheet with **F5**, with a band playing (any demo, or a Library style).
2. Right-click a section's name on the sheet. The section's bars are selected, and the menu has **Variation a**, **Variation b** and **Style: ...**, which names the style playing there now.
3. Open **Style**, then a genre, and pick a style, say **Latin** > **Bossa nova**. The bars of that section are now played as a bossa nova. The sheet writes "Bossa nova" after the number of the first bar, and the notice says which bars changed.
4. The same **Style** menu is in every bar's right-click menu. Without a selection it changes the style from that bar on; with bars selected (click the first, Shift+click the last), it changes only those bars, and the bars after them keep what they played.
5. **The song's style**, at the top of the menu, goes back to the band's own style.



The menu only lists styles in the time signature of those bars, and the [Orchestra](#), which plays in any. A waltz in the middle of a song in 4/4 would need bars of three beats, which means changing the song's bars, not only how the band plays them. Changing the time signature is a separate step, so it stays separate here.

What changes and what stays

A section's style changes **how** the band plays:

- the piano's **comping**, the rhythm it strikes the chords in,
- the **bass line**,
- the **drums**,
- the patterns of the guitar, the strings and the organ, where the band has them,
- the **swing**: a swing section in a straight pop song swings its eighth notes, and a rock section in a jazz song plays them even.

It does not change **who** plays or the sounds they play with. The band keeps its instruments, so the same piano and the same bass go from the bossa nova verse to

the swing bridge, as a real band would. To change the instruments, use the band's **Instrument lineup**.

The drums tell the listener a change is coming, as they do for a variation: the bar before the change plays the fill of the style that is ending, and the first beat of the new style gets a crash cymbal. Each player's comping pattern starts again from its beginning on the first bar of the new style, so a two-bar bossa nova pattern does not start halfway through.

Markers, the same way as variations

```
pub struct StyleMarker {  
    pub start: usize,  
    pub style: Option<&'static BandStyle>,  
}
```

A marker with `style: None` goes back to the band's own style. The markers live in the `Band`, next to the variation markers, and follow the same rules: a `tidy` of their own keeps only markers that change something, and everything that moves the song moves them too.

`&'static BandStyle` is a reference that lives for the whole program: the band styles are built once (`band_styles()` keeps them in a `OnceLock`) and never freed. That is why a marker can hold a reference instead of a copy or a name. A file can't hold a reference, so `StyleMarkerFile` saves the style's name and looks it up again when it reads the song; a name this version does not know is an error that says so.

Every style plays the whole song, then each keeps its own bars

The simplest way to write a part with several styles is not to teach every piece of the band about sections. Instead, `write_band` asks each style that plays somewhere for the whole song, and keeps each note only where that style plays:

```

for style in sections::styles_played(&band.styles) {
    let playing = band.playing(style);
    let mut part = self.band_notes(&playing, player, &score);
    if let Some(following) = &following
        && playing.style.follows_the_song()
    {
        part = self.played_along(following, &playing, player, part,
&score);
    }
    notes.extend(sections::where_it_plays(part, &band.styles, style));
}

```

`band.playing(style)` is a copy of the band with the other style's patterns. `band_notes` writes one player's part for one style (the `if let` in the middle is the [orchestra](#) playing along with a score). `where_it_plays` keeps the notes that start where `style` plays, and cuts a note that would ring into the next style:

```

notes
    .into_iter()
    .filter(move |(_, note)| style_in(markers, note.start) == style)
    .map(move |(channel, mut note)| {
        if let Some(change) = next_change(markers, note.start)
            && note.start + note.length > change
        {
            note.length = change - note.start;
            note.trim = 0;
        }
        (channel, note)
    })

```

`move` makes each closure own what it uses (`markers` and `style`, both cheap to copy), so the iterator can be returned from the function and used after it ends. A song without section styles has one style, `None`, and keeps every note.

The fills come from the same idea. The drums play a fill in the bar before a change of variation, and a change of style is a change too. When the band plays the outgoing style through the whole song, that style has a fill before the change, and the bar before the change belongs to it, so its fill is kept. The incoming style has a crash on the change, and the change is its first bar, so its crash is kept.

Swing per section

Swing is played, not written: notes stay on the grid, and playback moves the second eighth note of each beat later (see [Swing](#)). A band pattern in a section with its own style swings as that style does:

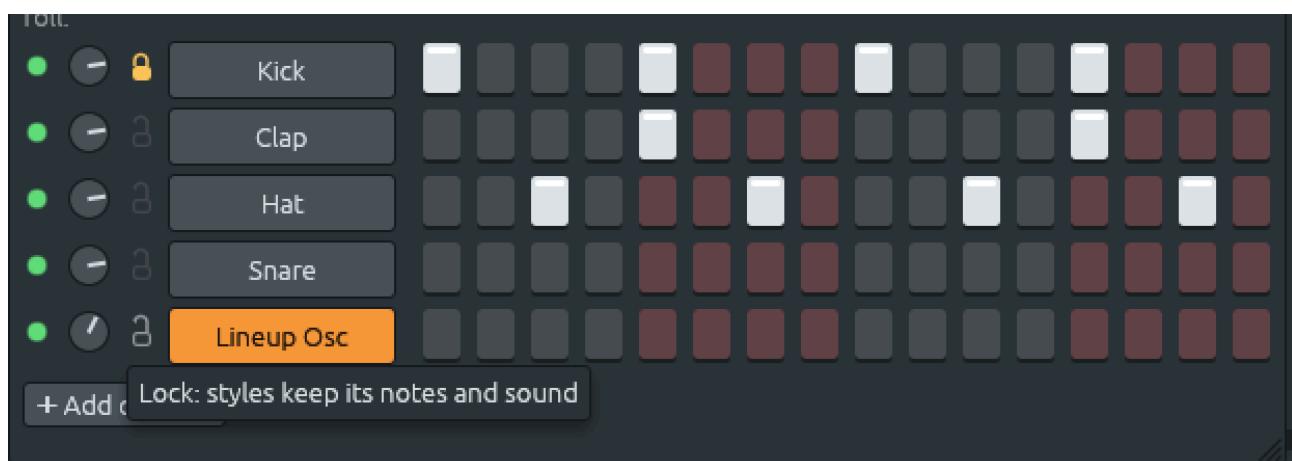
```
pub fn pattern_swing(&self, pattern: usize) -> u8 {
    let swing = self.section_swing(pattern).unwrap_or(self.swing);
    if swing > 0 && swings_in(self.pattern_time(pattern)) {
        swing
    } else {
        0
    }
}
```

`section_swing` looks at where the pattern's first clip is and returns that section's swing, or `None` (the song's swing) for your own patterns and for sections in the band's own style. The band starts a new pattern where a section's style starts, and two stretches with the same notes but different swing never share a pattern, so each pattern swings only one way.

The [playback trace](#) shows it: in a straight Pop beat song whose bars 5 to 8 are Swing, the ride cymbal's and the piano's offbeats in bar 5 start 2.667 sixteenths into the beat, the last third of a triplet, while in bars 1 to 4 they are on the grid.

Locking a track: styles play around it

Say you played a piano part yourself, or picked a nylon guitar for the chords, and now want to hear the song as a bossa nova. Picking the style would rewrite every track it plays: your piano part would be replaced by the style's comping, and your guitar would go back to the style's instrument. So you can **lock** a track first. A locked track keeps its notes and its instrument, and the style plays the rest of the song around it.



Trying it

1. Open a demo, for example **Ode to Joy**, and show the rack with **F6**.

2. Click the small padlock between **Keys**'s knob and its name. It closes and turns orange. Right-clicking the name has **Lock** and **Unlock** too, and **Ctrl+Z** undoes it.
3. In the **Library**, apply another style, say **Bossa nova**. The band comes in on its own tracks, but leaves the piano to your Keys: the notice ends with  **Kept: Keys.**, the Library's list of where each part came from has a **Locked** row, and the chord sheet's band says " Keys is locked: the band leaves the piano to it."
4. Try styles one after another in pattern mode. Each trial copies the locked rows into the trial pattern, so the new bass follows the locked chords.
5. Unlock Keys again. The band takes the piano part back, and the notice says your Keys is muted so the two do not play over each other.

What moves the *whole* song still moves a locked track with it: a new time signature puts its notes in the new bars, and a key change moves them to the new key. A lock protects a part from styles, not from the song changing shape. You can always edit a locked track by hand.

One bit per channel

The rack has at most 32 channels, so the set of locked ones fits in one `u32`, one bit for each channel:

```
pub struct Locked(u32);

// One bit for each channel there can be.
const _: () = assert!(MAX_CHANNELS <= u32::BITS as usize);

impl Locked {
    pub fn contains(self, channel: usize) -> bool {
        channel < MAX_CHANNELS && self.0 & (1 << channel) != 0
    }
}
```

`Locked(u32)` is a **tuple struct**: a new type wrapped around a `u32`, reached with `.0`. Wrapping it means nobody can add two sets by mistake or pass a channel count where a set is meant. The `const _` line is a check the compiler runs: if someone raised `MAX_CHANNELS` above 32, the build would fail right there instead of a lock silently landing on the wrong channel.

Removing a channel renumbers the ones after it, so the set must shift too. `without` keeps the bits below the removed channel and moves the bits above it down by one:

```
let below = self.0 & ((1 << index) - 1);
let above = self.0.checked_shr(index as u32 + 1).unwrap_or(0) << index;
Locked(below | above)
```


`checked_shr` returns `None` instead of overflowing when the shift is 32 or more, which happens when the last channel is removed.

One door for every style

Styles write into patterns from many places: a whole style, a rhythm, a saved style, a row or micromode style, a fill, the final chord. If each of them had to remember to skip locked channels, one would forget. So the library never gets a `&mut Pattern` at all. It gets a `StylePattern`, which only writes rows that are not locked:

```
pub(crate) struct StylePattern<'a> {
    pattern: &'a mut Pattern,
    locked: Locked,
}

impl StylePattern<'_> {
    pub(crate) fn set_notes(
        &mut self,
        channel: usize,
        notes: impl IntoIterator<Item = NoteEvent>,
    ) -> bool {
        !self.locked.contains(channel) && self.pattern.set_notes(channel,
        notes)
    }
}

impl Deref for StylePattern<'_> {
    type Target = Pattern;

    fn deref(&self) -> &Pattern {
        self.pattern
    }
}
```

`Deref` lets a `StylePattern` be read exactly like a `Pattern` (`pattern.length()`, `pattern.notes(channel)`), while writing only goes through its own methods. This is a common Rust shape: a **wrapper** that hands out reading for free and guards writing. The instrument has the same kind of door, `Project::set_sound`, which refuses a locked channel. A test applies every style, rhythm, saved style, row and micromode style with each channel locked in turn, and checks that the locked one never changes.

The band plays around a lock

The band writes its own patterns, a part per player, so it has its own rules (in `band/locks.rs`):

- a player whose channel is locked is skipped: its part stays as it is;
- locked drum channels keep their notes while the other drums change;
- if your own Keys or Bass is locked and plays, the band's piano or bass stays silent, so the song's part is not doubled;
- when the band leaves (**No band**), a locked band part stays in the song as your own.

The band keeps a fingerprint of what it plays from, to know when to write again (see [A part you edit stays yours](#)). The lock counts in that fingerprint only when something is locked, so a song with nothing locked keeps the fingerprint it had.

In the file

All four are saved with the band, and each is left out of the file when it is not used, so a song that does not use them is written as it would be without them:

```
#[serde(default, skip_serializing_if = "Form::is_plain")]
pub form: Form,
#[serde(default, skip_serializing_if = "Vec::is_empty")]
pub variations: Vec<VariationMarker>,
```

`Form::is_plain` is `true` for the form a song has without the form row: no intro, one chorus, no ending. A song that uses the form needs at least format 14, variations format 15, a locked channel format 23, and section styles format 24 (see [The project file format](#)).

Testing it

Each module tests its own part:

- `form.rs`: `choruses_play_the_sheet_again_and_again` checks the playlist shows the chorus once and the arrangement plays it three times (192 steps for a 64-step chorus), with "2 of 3" at step 70; `an_intro_leads_into_the_chorus` and `the_intro_moves_the_melody_along` put intros in and take them out, and check the melody moves and the band's parts are not frozen; `an_intro_is_a_turnaround_into_the_first_chord` checks turnarounds into major and minor chords, the spellings into Ebm and Gb, and a blues; `the_ending_holds_the_tonic` checks the last bar is one chord held 16 steps, with one crash; `the_form_is_saved` checks the file reads back the same.

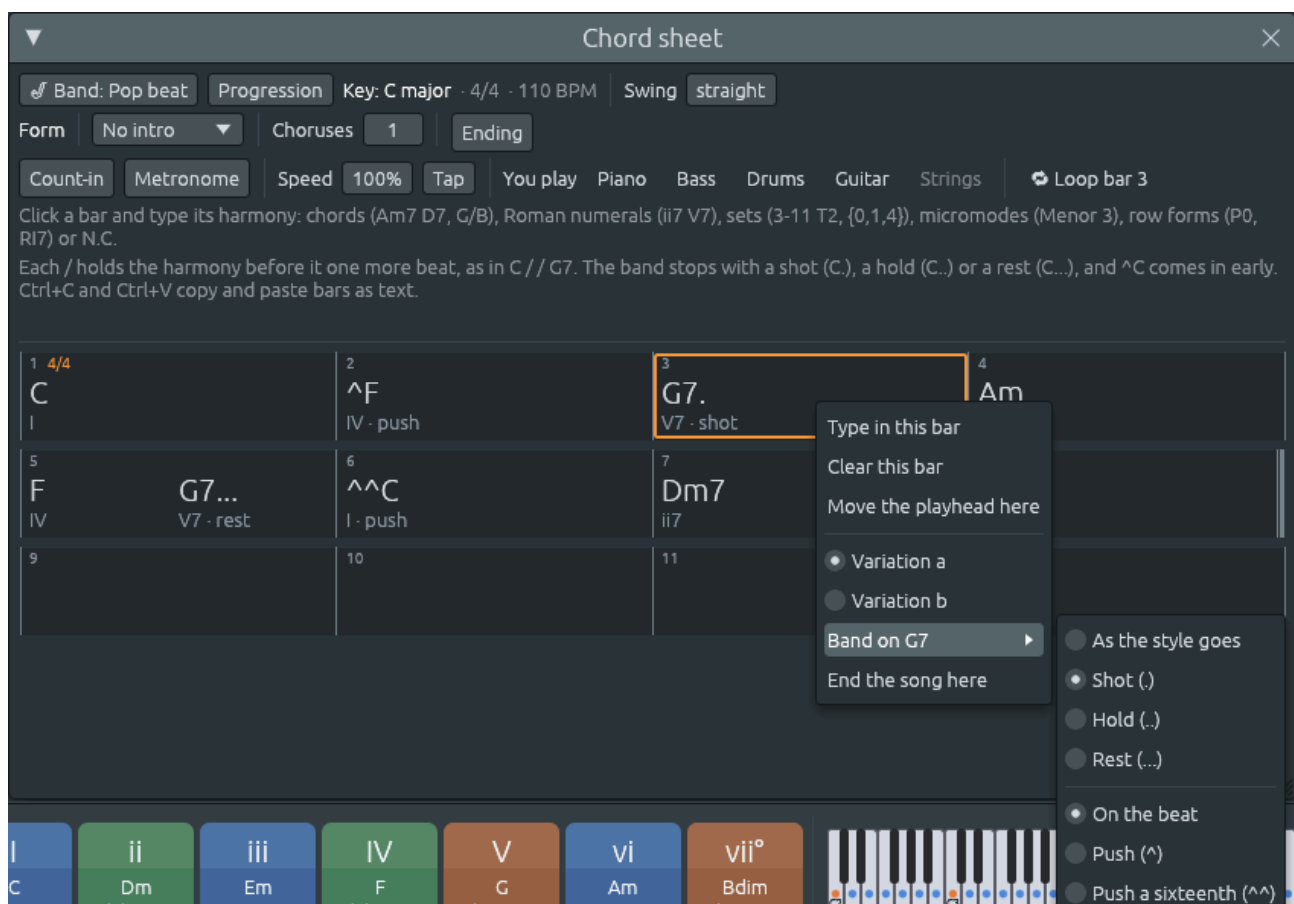
- `variation.rs`: `markers_keep_only_changes`, `the_drummer_moves_to_the_ride`, `the_drums_fill_into_a_change`, `variation_b_is_louder`, `players_switch_to_their_busier_way`, `a_jazz_ballad_walks_in_variation_b`, `markers_move_with_the_song` and `a_copied_section_plays_the_same_variations`.
- `sections.rs`: `a_section_plays_as_its_style_does`, `a_section_plays_its_drums_and_starts_its_pattern_on_its_first_bar`, `a_section_without_drums_rests_them`, `a_swing_section_swings_in_a_straight_song` and `markers_move_with_the_song_and_are_saved`.
- `lock.rs`: `no_library_choice_writes_a_locked_channel` is the test that tries every style with each channel locked; `removing_a_channel_renumbers_the_ones_after_it` checks `without`. `band/locks.rs` checks each rule of the band, and that `no_band_leaves_the_locked_parts_to_the_song`.

In the app, `ux_tests.rs` clicks the rack's lock and undoes it (`the_rack_locks_a_channel_and_undo_unlocks_it`), tries Doo-wop and then Bossa nova in a pattern around a locked Keys, and checks the notices and the chord sheet say what is kept (`the_library_and_the_band_say_what_is_kept`).

Shots, holds, rests and pushes

Files: `crates/lineup-core/src/chord_sheet.rs`, `crates/lineup-core/src/band/stops.rs`, `crates/lineup-core/src/band.rs`, `crates/lineup-core/src/file.rs` and `crates/lineup-app/src/chord_sheet.rs`

A style plays the same groove bar after bar, and that is what makes it a style. But listen to a real band: at the end of a phrase they all hit one chord together and stop dead, or hold a chord out under the singer, or drop out for a bar so the drummer or the soloist is alone, or come in with the next chord a little before the beat. Arrangers write these moments into the chart, and they are what make a band sound arranged rather than looped. The chord sheet has marks for them, and the demo songs' charts use the same marks.



Trying it

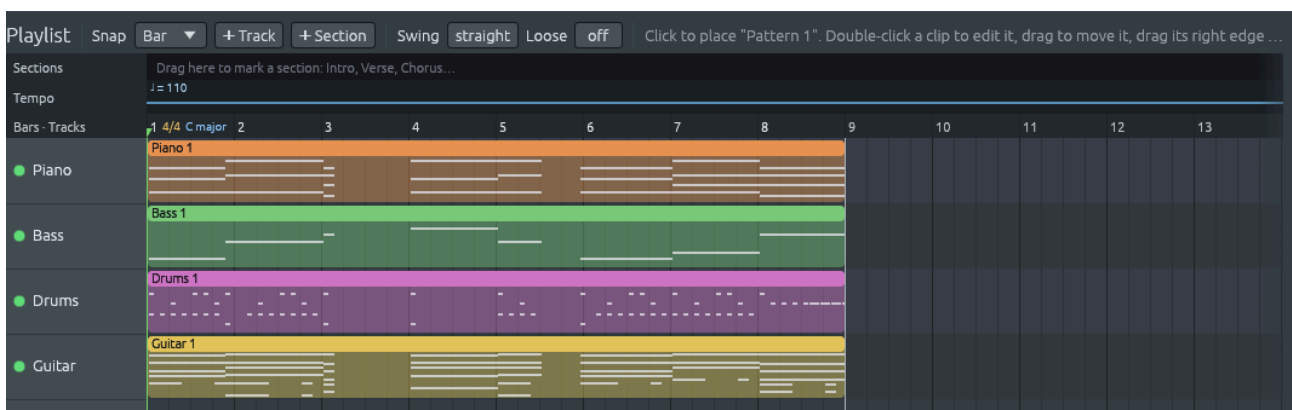
Press **Ctrl+N** for a new song with a band, and type these bars on the chord sheet, one per bar:

| C | ^F | G7. | Am.. | F G7... | ^^C | Dm7 | G7 |

Under each marked harmony, a word says what the band does there: **push**, **shot**, **hold** or **rest**. Hover over one for the whole sentence, as in “Hold: the band strikes it and holds it until the next harmony”. Press Space and listen:

- bar 2: the band plays F on the “and” of beat 4 of bar 1, an eighth note early, and lets it ring over the bar line, where it is written;
- bar 3: the whole band strikes G7 once, short, with a kick and a crash, and then nothing until bar 4;
- bar 4: Am struck and held for the whole bar;
- bar 5: F in the style for two beats, then silence for G7;
- bar 6: C a sixteenth note early;
- bars 7 and 8: the style again.

The playlist shows the same: the band’s parts have gaps and long notes where the marks are.



The marks can also be set without typing: right-click a bar and pick **Band on G7** (one item for each harmony in the bar), then a stop or a push. **As the style goes** and **On the beat** take the marks off. Like every change on the sheet, the band plays again, undo takes it back, and the marks are copied with the bars, as text too (see [Chord sheets as text](#)):

| C. | ^F | G7.. Am... | ^^C |

The music: breaking the groove

Shots

A **shot** (also called a hit, a stab or a kick) is a chord the whole band plays at once, short and accented, followed by silence. Big band charts are full of them: the brass hits a chord, and the drummer plays it with the band on the crash cymbal and the bass drum, often with a short fill before it to “set it up”. A run of shots, one at the start of each bar or two, is called **stop-time**: the band plays only the hits and leaves the rest of the bar to the singer, the soloist or a tap dancer. In the verses of Muddy Waters’ “Hoochie Coochie Man” the band plays only a short riff at the start of each bar and the voice fills the silence; it is one of the best-known stop-time grooves in the blues.

Here a shot is an eighth note. Every player strikes the chord its own way (the piano its voicing, the bass the bass note, the guitar a strum), a little harder than the style, and the drums play a kick and a crash. Then the band is silent until the next harmony. That is how bar 3 above works: the silence is part of the music, and the harmony is still G7 for whoever plays over it.

Holds

A **hold** is a chord struck and held. Bands use it at the end of a phrase, for a big chord under the last word of a line, or to let a soloist play freely over one harmony. It is not a fermata: a fermata stops the beat, and the band waits for the conductor, while a hold lasts exactly until the next harmony, and the beat goes on underneath. Each player strikes the chord with a kick and a crash, as for a shot, and lets it ring. A row form, which the band plays as trichords one after another, is held one trichord at a time: each is struck as it comes and held until the next.

Rests

A **rest** is the band dropping out. Arrangers call it a **break**: in jazz, the band stops for a bar or two and the soloist plays alone (the famous “break” at the end of a tune’s first chorus), and in funk and hip hop the bars where only the drummer plays gave the “breakbeat” its name. Here the whole band rests, drums too; write the drums’ break yourself in the piano roll, or play it with **You play** (see [Practice tools](#)).

Pushes

A **push** (or **anticipation**) is a chord played before the beat it belongs to and held over it. In C | ^F, F is written on beat 1 of bar 2 but played on the “and” of beat 4 of bar 1, and the band does not strike again on beat 1. A chart writes it as an eighth note tied over the bar line:

bar 1				bar 2			
1	2	3	4 &		1	2	3 4
C	C	C	C F---		-F	F	F F

^ the F is struck here, and held over beat 1

Without the push, the downbeat is the strongest moment of the bar, and everybody expects it. With it, the chord comes early and beat 1 is silent: the music leans forward, as if it could not wait. That is **syncopation**, an accent off the beat, and pushes are everywhere in pop, rock, funk and Latin music. Salsa bass players anticipate nearly every chord: in the **tumbao**, the bass plays the next chord’s root on beat 4, held over the bar line. A push of a sixteenth note (^^C) is subtler, the pull of funk and fast Latin music.

A push can be combined with a stop: ^G7. is a shot an eighth early.

Why dots and carets

Dots after a harmony and carets before it are a short way to write the marks without leaving the keyboard: one dot for the shortest (a shot), three for the longest silence (a rest). The caret ^ looks like the **marcato**, the accent sign musicians write over a note to play it strongly, and a push is played with an accent. N.C. (no chord) keeps its own dots: they are part of its name.

How it works

Two optional marks on a marker

A harmony on the sheet is a `ChordMarker`, with two optional fields, `push` and `stop`, `None` when it is not marked (see [How the sheet is stored](#)). Their types are two small enums:

```
pub enum Stop {
  /// "C.": the whole band strikes the chord once, short, with a kick
  /// and a crash cymbal, and then is silent.
  Shot,
  /// "C..": the band strikes the chord and holds it.
  Hold,
  /// "C...": the band is silent. The harmony goes on, for whoever
  /// plays over it.
  Rest,
}
```

```
pub enum Push {
  /// "^C": an eighth note early.
  Eighth,
  /// ^^C": a sixteenth note early.
  Sixteenth,
}
```

Each knows how to write itself (`marks` : `"."`, `".."`, `"..."`, `"^"`, `"^^"`), how many steps early a push is (`steps` : 2 or 1), and a sentence for the hints (`describe`). The two fields are left out of the file when they are `None` , so a song without marks is written as it would be without them. The file's format only becomes 18 when some marker has a mark, checked with `any` :

```
let has_breaks = playlist
  .chord_markers()
  .iter()
  .any(|marker| marker.push.is_some() || marker.stop.is_some());
```

`ChordMarker::pushed` and `ChordMarker::stopped` take a marker by value and give it back changed, so a marked marker reads like a sentence: `ChordMarker::new(16, f).pushed(Some(Push::Eighth))` .

Reading the marks

Typing `^^Am7/G..` must give a push, the harmony `Am7/G` and a stop. `read_marks` peels the marks off both ends and counts them:


```

let name = text.trim_start_matches('^');
let push = match text.len() - name.len() {
    0 => None,
    1 => Some(Push::Eighth),
    2 => Some(Push::Sixteenth),
    _ => {
        return Err(
            tr("a push is written ^ (an eighth early) or ^^ (a
sixteenth).").to_string(),
        );
    }
};
if is_no_chord(name) {
    return Ok((push, name, None));
}
let bare = name.trim_end_matches('.');
let stop = match name.len() - bare.len() {
    0 => None,
    1 => Some(Stop::Shot),
    2 => Some(Stop::Hold),
    3 => Some(Stop::Rest),
    _ => {
        return Err(tr(
            "after a harmony, . is a shot, .. a hold and ... a rest: no
more dots.",
        )
        .to_string());
    }
};
Ok((push, bare, stop))

```

`trim_start_matches` returns a **slice** of the same string without the carets, not a copy, so the difference of the two lengths is how many it took off. A caret and a dot are one byte each in UTF-8, so counting bytes counts marks. A no-chord (`N.C.` , `NC`) is left whole, or its last dot would become a shot. The rest, `bare` , goes to the parser that reads chords, numerals, sets and rows.

Writing a marker back is the same in reverse, `format!` with the marks on both sides:

```

pub fn text(&self) -> String {
    let carets = self.push.map_or("", Push::marks);
    let dots = self.stop.map_or("", Stop::marks);
    format!("{carets}{{{dots}}}", self.harmony)
}

```

`map_or("", Push::marks)` gives `""` for `None` and calls `Push::marks` on the value inside otherwise. A method can be passed by name like this, as a function that takes the value as its first argument.

After the style: rewriting each part

The players do not need to know about marks. Each style writes its part as it always does, and then a last step, `Breaks::apply` in `band/stops.rs`, rewrites it. First `Breaks::of` turns the markers into stretches of the song:

- a **stop** lasts from its harmony (earlier if it is pushed too) to where the next harmony comes in (earlier if that one is pushed);
- a **push** lasts from where it is played to the next harmony. A push never reaches back to the harmony before it: if F is written one step after C, it cannot come in earlier than C.

Then, for each player, the stops:

```
for stop in &self.stops {
    clear(&mut notes, stop.start, stop.end);
    let Some(harmony) = harmony_at(stop.written) else {
        continue;
    };
    match stop.stop {
        Stop::Shot => notes.extend(accented(strike(&Played {
            start: stop.start,
            end: (stop.start + SHOT_STEPS).min(stop.end),
            ..harmony.clone()
        }))),
        // A row form is played as trichords one after another:
        // each is struck as it comes, and held.
        Stop::Hold => {
            // ...
        }
        Stop::Rest => {}
    }
}
```

`clear` takes out the notes that start in the stretch, and shortens the ones that started before it and would still be ringing, so that a shot is followed by real silence. `accented` raises every note to at least velocity 100, a little harder than the style. `SHOT_STEPS` is 2, an eighth note. A rest strikes nothing: its arm is empty.

A push clears the beat it is written on and strikes the harmony early. How long it is held depends on the player: up to where that player's part goes on. In the song above, the Pop beat piano plays eighth notes, so the pushed F is held from the "and" of beat 4 to the "and" of beat 1, a quarter note; a bass playing half notes would hold it to beat 3.

Each player strikes its own way

What does the piano strike, and what does the bass? `apply` does not know; it takes a closure that does:

```
pub(crate) fn apply(
    &self,
    mut notes: Vec<NoteEvent>,
    harmonies: &[Played],
    strike: impl Fn(&Played) -> Vec<NoteEvent>,
) -> Vec<NoteEvent> {
```

The piano passes its held chords, the bass a function that plays one note, and the guitar and strings check first whether they are playing in this variation at all (strings that only come in for variation b should not suddenly hit a shot in variation a):

```
score.breaks.apply(notes, &score.harmonies, |played| {
    match extras.guitar_in(variation_at(played.start)) {
        Some(_) => guitar::play(
            GuitarPart::Strum(Comping::Held),
            Comping::Held,
            &score.bars,
            std::slice::from_ref(played),
        ),
        None => Vec::new(),
    }
})
```

This closure uses `extras`, `variation_at` and `score`, variables of the function around it. A closure that **captures** its surroundings carries them with it, so it is not a plain function pointer like the `fn(&Played) -> Vec<Note>` that `strummed` takes (see [Enums that carry data, and a function as a parameter](#)), and its type has no name you could write. `impl Fn(&Played) -> Vec<NoteEvent>` says “any type that can be called like this”, and the compiler writes a version of `apply` for each closure passed to it. A plain function fits too: the bass passes `bass_struck` by name.

`std::slice::from_ref(played)` turns one `&Played` into a slice of one, `&[Played]`, without copying it, because `guitar::play` takes a slice.

Why does the bass not use its own held style, `Bass::Held`? Because that one strikes again on every bar line, and a hold over two bars would be struck twice. `bass_struck` plays one note for the whole stretch.

The drums

The drums are different: their notes come in pairs, `(channel, note)`, one channel per drum. `apply_to_drums` takes the groove out of each stretch and adds the hits, on the kick's and the crash's channels:

```
notes.retain(|(_, note)| !(stop.start..stop.end).contains(&note.start));
```

`(a..b).contains(&x)` reads better than `a <= x && x < b`, and for a push, `(push.start..=push.written)` with `..=` includes the step it is written on. The kick and the crash must have channels before they can play: a style without a crash gets one, only when the sheet has a mark, through

```
.chain(
  (!breaks.is_empty())
    .then_some([Drum::Kick, Drum::Crash])
    .into_iter()
    .flatten(),
)
```

`then_some` turns a `bool` into an `Option: Some([...])` when true, `None` when false. `into_iter().flatten()` then gives the two drums or nothing, so the list of drums the style needs grows only when needed.

The band plays again when a mark changes

No code is needed for that. The band remembers a fingerprint of everything it plays from, including the chord markers, made from their `Debug` text (see [A part you edit stays yours](#)). A marker's `Debug` text includes its marks, so changing a shot into a hold changes the fingerprint, and the band plays again. Parts you edited by hand stay frozen until **Play it again**, as always.

Testing it

The tests in `band/stops.rs` work on `Breaks` alone, with a few `Played` harmonies made by hand: `marks_become_stretches_up_to_the_next_harmony`, `a_push_never_reaches_back_past_the_harmony_before`, `a_shot_strikes_once_and_the_band_stops`, `a_hold_last_until_the_next_harmony_and_a_rest_is_silent`, `a_hold_over_a_row_form_holds_each_trichord_as_it_comes`,

`a_push_comes_in_early_and_is_held_over_the_beat` and `the_drums_stop_and_hit_with_the_band`. In `chord_sheet.rs`, `marks_tell_the_band_where_to_stop_and_push` reads every mark, on chords, numerals and micromodes, `N.C.` with and without a push, and the mistakes (`C....`, `^^^C`, a lone `^`); `marks_are_written_back_as_typed` writes them back. `the_band_stops_and_pushes_where_the_sheet_says` in `band.rs` plays a song with marks and checks every player, and `shots_holds_rests_and_pushes_need_format_eighteen` in `file.rs` checks that only the marked harmonies get `push:` and `stop:` in the file.

To hear it without listening, save the song above and read its [playback trace](#): the piano's F starts in bar 1 on beat 4, two steps into the beat; in bar 3 the piano strikes G7 once, an eighth long; in bar 4 its Am lasts the whole bar; and the pushed C starts in bar 5 on beat 4, three steps in.

The soloist

Files: `crates/lineup-core/src/harmony/chord_scale.rs`, `crates/lineup-core/src/band/solo.rs`, `crates/lineup-core/src/band.rs`, `crates/lineup-core/src/file.rs`, `crates/lineup-app/src/chord_sheet.rs` and `crates/lineup-app/src/piano_roll.rs`

The band accompanies: it plays the chords so that someone else can play the tune or improvise over them. The **soloist** is that someone else, a player who improvises a line over the chord sheet. It is experimental and off until you switch it on, because a solo made by rules is a study, not a performance. But it is a useful study: every note it plays has a reason, and the piano roll shows the reason, so you can see how an improvised line is built and try the same rules on your own instrument.

Trying it

Press **Ctrl+N** for a new song with a band, and type these bars on the chord sheet, one chord a bar:

```
| Dm7 | G7 | Cmaj7 | A7 | Dm7 | G7 | C6 | C6 |
```

In the **Library**, apply **Swing** (under Jazz). Then open the sheet's  **Band** menu and tick **Soloist (experimental)**. Hover over it first: the hint says who the soloist is in this style, a tenor sax. A **Soloist** track appears in the playlist, and the channel rack gets a Soloist channel with the tenor sax. Press Space: over the band, the sax plays two-bar phrases in swung eighth notes, with a breath between them.

Another solo in the same menu plays a different one; undo brings the last one back. Untick the soloist and the track and the channel go away. Like every band part, the solo is played again when the sheet changes, and if you edit its notes it stays as you left it until you press **Play it again** (or **Another solo**).

Now double-click the soloist's clip to open its part in the piano roll. Its notes are coloured by what they are:

- **green** for chord tones, marked with which one: **R** for the root, **3**, **5**, **7** (and **6**, **9** when the chord has them), or **•** for a note of a set, which has no root to count from;

- **blue** for the other notes of the chord's scale: **p** a passing note, **n** a neighbour note, **s** any other;
- **orange** for chromatic notes, from outside the scale: **a** an approach, **cp** a chromatic passing note.

Point at a note, or play the song, and the line above the grid says what it is, in a sentence:

Piano roll · Soloist 1 · Soloist

Key: C major C D E F G A B Snap to scale Row... Sets... Micromodes... Draw: Single note Root Length 1 beat Tools

✓ Ghost notes

Chords in C major: I - C ii - Dm iii - Em IV - F V - G vi - Am vii* - Bdim

Point at or hold a chord to see what often follows.

Bar 3, beat 4, step 3 - Bb3: a chromatic passing note, a half step from the notes either side. Over Cmaj7 the scale is C major (Ionian): C D E F G A B.

Soloist: chord tones green (R 3 5 7), scale notes blue (p passing, n neighbour, s other), chromatic orange (a approach, cp passing)

Soloist: B C B A G F F# G A B D E F G C B Bb A G E F G A

All: Dm7 >->Dm7/f F6 Dm7 A-G Dm7 F# G7 >-> G7 G7/B G7 D G7 B Cmaj7 >-> Cmaj7/E Cmaj7 G-C Cmaj7 Bb A7 A A7 >7/C# A7 E A7 Db Dm7 >->Dm7/f F6 Dm7

Velocity

In this solo, bar 3 ends with B3 (the 7th of Cmaj7) on beat 4, then B $\flat$ 3 on the “and”, then A3 on beat 1 of bar 4, the root of A7: three notes a half step apart, walking down into the next chord. The B $\flat$ is in neither chord, and that is fine: it is on the weak half of the beat and it leads somewhere.

The music: how an improvised line is built

Chord tones on the beat

Ask a jazz teacher how to start improvising over changes, and the first rule is nearly always the same: **play chord tones on the strong beats, and anything that connects them in between**. On the beat the ear checks the melody against the harmony; between the beats it is moving and forgives more. A line that lands on a chord tone on each beat “makes the changes”: you can hear the chords in it even without the band.

The most telling chord tones are the **3rd** and the **7th**, the **guide tones**. They say whether a chord is major or minor and what kind of seventh it has, and from one

chord to the next of a ii–V–I they move by half steps: the 7th of Dm7 (C) falls to the 3rd of G7 (B), and the 7th of G7 (F) falls to the 3rd of Cmaj7 (E). Many solos are guide tones joined up.

The soloist chooses, for each beat, the chord tone **nearest** to where it is going: the line moves up or down, and keeps its direction until it turns at the edge of the instrument's range, or now and then on its own. One time in five it skips a chord tone and leaps to the next. That gives lines that mostly move by small steps, as tunes do, with a leap here and there.

Chord scales

Between the chord tones go other notes, and which ones depends on the chord: over Dm7, is the note between A and C a B or a B $\flat$? Jazz teaching answers with **chord scales**: each chord gets a seven-note scale that contains it, and the notes between come from that scale.

The scale is not chosen from the chord alone, but from the chord **in its key**. Over Dm7 in C major, the B is natural, because the music is in C: the scale is C major from D, D E F G A B C, called **D Dorian**. The modes of C major are what the chords of C major play:

Chord in C major	Its scale	Notes
Cmaj7 (I)	C Ionian	C D E F G A B
Dm7 (ii)	D Dorian	D E F G A B C
Em7 (iii)	E Phrygian	E F G A B C D
Fmaj7 (IV)	F Lydian	F G A B C D E
G7 (V)	G Mixolydian	G A B C D E F
Am7 (vi)	A Aeolian	A B C D E F G
Bm7 $\flat$ 5 (vii)	B Locrian	B C D E F G A

Seven names for the same seven notes. The names matter because they say which note is the root and where the half steps fall from it: Dorian is a minor scale with a major 6th (B over D), Lydian a major scale with a raised 4th (B over F), Mixolydian a major scale with a minor 7th (F over G). Those are the notes that give each chord its colour in the key.

Other chords get their scale the way a player thinking in the key would choose it, in this order:

1. **A chord of the key** plays the key's scale from its root, as in the table. In a minor key, a chord that uses the raised 7th (E7 in A minor, with G#) plays the key's **harmonic minor** from its root: A harmonic minor from E, E F G# A B C D, called **E Phrygian dominant**.
2. **A chord borrowed from the parallel key** plays that key's scale: Fm in C major comes from C minor, so it plays C minor from F, which is F Dorian. Bb7 in C major (from C minor too) plays Bb Mixolydian.
3. **A secondary dominant** plays the scale of the chord it leads to. D7 in C major is the V7 of G, so it plays G major from D, D Mixolydian. A7 in C major is the V7 of Dm, a minor chord, so it plays D harmonic minor from A: A Bb C# D E F G, A Phrygian dominant. Over A7, a Bb is a scale note, pointing to the D minor chord that follows.
4. **Anything else** plays the scale its kind usually gets: **Lydian** for a major chord from outside the key (the Db of the Neapolitan sixth in C), **Lydian dominant** for a dominant seventh that comes from outside and leads nowhere, **half-whole diminished** for 7b9 and 7#9, **whole-half diminished** for a diminished seventh, **whole tone** for an augmented chord, **Locrian** for a half-diminished one, **Dorian** for a minor chord. In a key that is not a seven-note scale (a blues scale, a pentatonic one) every chord gets the usual scale of its kind, with major chords on plain major (Ionian) and dominants on Mixolydian.

One more rule: **the chord's own notes win**. E9 in C major leads to Am, so it starts from A harmonic minor (E F G# A B C D), but its 9th is F#, and F and F# cannot both be in the scale with the same letter. The F# of the chord replaces the F: E F# G# A B C D, **Mixolydian b6**.

There is a second school, which gives every chord its own scale without asking about the key (Dorian for every m7, Mixolydian for every 7). It is simpler to teach and often sounds the same. Thinking in the key is closer to how the tunes were written, and it keeps the notes that chords of the key have in common the same from chord to chord.

Notes between the beats

Each kind of note between the chord tones has a name, and most of them are older than jazz: harmony textbooks call them **non-chord tones**.

- A **passing note** fills the gap between two chord tones a third apart, by step, going the same way: C **D** E over C.
- A **neighbour note** (or auxiliary note) steps away from a chord tone and comes back: E **F** E.
- A **chromatic passing note** fills a whole step with the half step between: D **D#** E.

- A **chromatic approach** comes into a chord tone from a half step below (or above): **F#** G. When the soloist starts a phrase with a note just before the beat, a **pickup**, it is usually one of these.
- Any other scale note, reached or left by a leap, is marked **s**, and a note from outside the scale that does none of this is marked **x**. The soloist does not play **x** notes, but a part you edit can have them.

A step here is a half step or a whole step, or, in harmonic minor, the augmented second between its 6th and 7th notes (F and G# in A), because those two are next to each other in the scale.

Chromatic passing notes on the “and” of the beat are part of the sound of **bebop**, the jazz of Charlie Parker and Dizzy Gillespie: a half step added between two scale notes keeps the chord tones on the beats when a line runs in eighth notes. Jazz teachers later wrote this down as the “bebop scales”.

Phrases and breathing

A singer has to breathe, and a horn player too, and so lines come in **phrases**. The soloist plays two-bar phrases. Each one may start a beat late or with a pickup, and leaves a beat or two empty at the end. A line that never stops sounds like an exercise; the rests are what make it sound like someone talking. Then the eighth notes are swung as the band is (see [Swing](#)), and the library’s phrasing for melodies shapes its loudness, as it does for the demos’ tunes.

The soloist plays over the chorus, not over the intro or the ending (see [The form](#)), and it keeps playing through shots and rests (see [Shots, holds, rests and pushes](#)): the band stops precisely so that a soloist can be heard alone. Over “no chord” it is silent. Over a pitch-class set, a micromode or a row form there is no scale to choose, and the soloist plays the set’s own notes, on the beats and between them.

Who plays

Each family of styles has its soloist, as a General MIDI instrument and the range it plays in. The family is the one of the band’s instruments (its **Instrument lineup**):

Styles	Soloist	Range
Pop	alto sax	G3 to G5
Rock	overdriven guitar	A3 to C6
Blues	harmonica	C4 to C6

Styles	Soloist	Range
Jazz	tenor sax	D3 to F5
Soul, funk and R&B	alto sax	G3 to G5
Hip hop and lo-fi	vibraphone	C4 to C6
Electronic	saw lead	C4 to C6
Latin	trumpet	B $\flat$ 3 to A5
Tango	tango accordion	G3 to C6
Latin American folklore	pan flute	E4 to E6
Flamenco	nylon guitar	E3 to A5
Balkan and Turkish	clarinet	D3 to B $\flat$ 5
Reggae and ska	trumpet	B $\flat$ 3 to A5
Country and folk	fiddle	G3 to E6
Classical and film	flute	F4 to F6

The tango accordion is General MIDI's stand-in for the bandoneon. The ranges are the comfortable middle of each instrument, not all of it: solos sit where an instrument sings.

How it works

Choosing a chord scale

`chord_scale` in `harmony/chord_scale.rs` follows the four rules above, one `if` after another:

```

pub fn chord_scale(chord: Chord, key: Scale) -> PitchSet {
    if !key.kind.is_tonal() || !key.is_heptatonic() {
        return usual_scale(chord, false);
    }
    let harmonic = |root: PitchClass| Scale::new(root,
ScaleKind::HarmonicMinor);
    let source = if key.is_chord_of_key(chord) {
        match key.uses_raised_seventh(chord) {
            true => harmonic(key.root),
            false => key,
        }
    } else if let Some(parallel) = key.borrowed_from(chord) {
        match parallel.is_diatonic(chord) {
            true => parallel,
            false => harmonic(key.root),
        }
    } else if let Some(target) = key.secondary_dominant_target(chord) {
        match target.quality.is_minor() {
            true => harmonic(target.root),
            false => Scale::new(target.root, ScaleKind::Major),
        }
    } else {
        return usual_scale(chord, true);
    };
    with_chord_notes(source, chord)
}

```

Almost all of the music theory is elsewhere: `is_chord_of_key`, `borrowed_from` and `secondary_dominant_target` are what the chord sheet uses to name borrowed chords and secondary dominants (see [The key's helpers](#)). This function only puts them in the order a player would ask the questions.

Three bits of Rust worth noticing:

- An `if` is an **expression** in Rust: `let source = if ... { a } else if ... { b } else { ... };` gives `source` the value of whichever branch ran. The last branch does not give a value; it `return s` from the whole function, which is allowed because `return` has the type `!` ("never"), which fits anywhere.
- `else if let Some(parallel) = key.borrowed_from(chord)` both tests and unpacks: the branch runs only when there is a parallel key, and inside it `parallel` is that key.
- `harmonic` is a **closure**, a small function written in place, used three times. `|root: PitchClass| Scale::new(...)` takes a root and gives a scale.

`usual_scale` is one `match` on the chord's quality, with a guard for the chords from outside the key:

```
Major | Add9 | Major6 | Six9 | Major7 | Major9 | Major13 if outside =>
LYDIAN,
Major | Add9 | Major6 | Six9 | Major7 | Major9 | Major13 | Sus2 => IONIAN,
```

and ends by joining the scale and the chord. A `PitchSet` is twelve bits in a number, one for each pitch class, so joining two sets is a bitwise or of the numbers:

```
PitchSet::from_bits(scale.bits() | chord.pitch_set().bits())
```

The chord's notes win, letter by letter

```
fn with_chord_notes(source: Scale, chord: Chord) -> PitchSet {
    let names = source.chord_notes(chord);
    let mut set = chord.pitch_set();
    for pitch_class in source.pitch_classes() {
        let letter = source.spell(pitch_class).letter;
        let replaced = names.iter().any(|name| {
            name.letter == letter
            && name.pitch_class() != pitch_class
            && !source.contains(name.pitch_class())
        });
        if !replaced {
            set.insert(pitch_class);
        }
    }
    set
}
```

The set starts with the chord's notes, and each note of the source scale joins it unless a chord note has the **same letter** but a different pitch, from outside the scale. Letters, not pitches, decide which note is replaced: the F# of E9 replaces F, not G. That is why the scale is spelled with note names (`NoteName` , a letter and an accidental) before it is compared, and not just with numbers.

The last condition, `!source.contains(...)` , keeps enharmonic spellings from doing damage. Fdim7 in C major is spelled F A $\flat$ C $\flat$ E $\flat\flat$, and C $\flat$ and E $\flat\flat$ have the letters C and E: without the condition they would knock C and E out of the scale, although C $\flat$ is B and E $\flat\flat$ is D, both in C major already. The test `every_chord_scale_holds_its_chord` plays every kind of chord on every root in several keys and checks that each scale holds its chord and has six to nine notes.

Naming the scale

The names are a table of slices, the semitones above the root and the name:

```
pub(crate) const NAMES: &[(&[u8], &str)] = &[
    (IONIAN, "major (Ionian)"),
    (DORIAN, "Dorian"),
    (&[0, 1, 3, 5, 7, 8, 10], "Phrygian"),
    // ...
    (&[0, 1, 4, 5, 7, 8, 10], "Phrygian dominant"),
    // ...
];

pub fn chord_scale_name(scale: PitchSet, root: PitchClass) ->
Option<&'static str> {
    let relative = scale.relative_to(root);
    NAMES
        .iter()
        .find(|(steps, _)| PitchSet::from_semitones(steps) == relative)
        .map(|(_, name)| *name)
}
```

`relative_to(root)` turns the scale so that the chord's root is 0. Then it is the same set as one of the table's rows, or none. `&'static str` says the name lives as long as the program: it is text written in the code, so the function can hand it out without copying it.

What each note is

The kinds of notes are an `enum`, and one of them carries data:

```
pub enum NoteRole {
    ChordTone(Option<u8>),
    Passing,
    Neighbour,
    ChromaticPassing,
    Approach,
    ScaleTone,
    Outside,
}
```

`ChordTone(Some(3))` is the 3rd; `ChordTone(None)` is a note of a set, which has no root to count from. `note_roles` reads a line and gives a role for each note. For a note of the scale it looks at its neighbours, `before` and `after`, which are `None` at the start of a phrase or after a rest:

```

match (before, after) {
  (Some(before), Some(after))
    if before == after && is_step(before, pitch, over.scale) =>
  {
    NoteRole::Neighbour
  }
  (Some(before), Some(after))
    if (before < pitch) == (pitch < after)
      && before != pitch
      && pitch != after
      && is_step(before, pitch, over.scale)
      && is_step(pitch, after, over.scale) =>
  {
    NoteRole::Passing
  }
  _ => NoteRole::ScaleTone,
}

```

Matching on a **tuple** of two `Option`s checks both at once, and the `if` after a pattern, a **match guard**, adds a condition. The arms are tried in order, so the neighbour note (same note before and after) is found before the passing note. `(before < pitch) == (pitch < after)` compares two `bool`s: true when the line goes up both times or down both times, which is what makes a passing note pass.

`note_roles` reads any line the same way, the soloist's or one you edited, so the piano roll can paint every soloist note from it. Its colour is a `match` on an `Option<NoteRole>` with guards too:

```

let mut fill = match role {
  Some(role) if role.is_chord_tone() => NOTE,
  Some(role) if role.is_chromatic() => SOLO_CHROMATIC,
  Some(_) => SOLO_SCALE,
  None if scale.contains(note.pitch.pitch_class()) => NOTE,
  None => NOTE_OUTSIDE,
};

```

The `None` arms are what every other part gets: green in the key, red outside it.

Dice that always roll the same

An improvised solo needs chance: which chord tone, whether to leap, whether to start late. But the band must play the same notes every time the song is opened, or the parts you froze would no longer match the sheet. So the soloist rolls **pseudo-random** dice: a small function that turns a number, the **seed**, into a stream of numbers that look random but are always the same from the same seed.

```

struct Dice(u64);

impl Dice {
    fn next(&mut self) -> u64 {
        self.0 = self.0.wrapping_add(0x9E37_79B9_7F4A_7C15);
        let mut x = self.0;
        x = (x ^ (x >> 30)).wrapping_mul(0xBF58_476D_1CE4_E5B9);
        x = (x ^ (x >> 27)).wrapping_mul(0x94D0_49BB_1331_11EB);
        x ^ (x >> 31)
    }

    fn chance(&mut self, percent: u64) -> bool {
        self.next() % 100 < percent
    }
}

```

This is **SplitMix64**, a well-known generator: add a large odd constant, then scramble the bits with shifts, exclusive ors (`^`) and multiplications. In Rust an ordinary `+` or `*` that overflows stops a debug build with a panic, because overflow is usually a bug. Here it is the point, so `wrapping_add` and `wrapping_mul` say so: the result wraps around past the largest `u64`, as the algorithm expects. The underscores in `0x9E37_79B9_7F4A_7C15` are only for reading.

`struct Dice(u64)` is a **tuple struct**: a struct with one unnamed field, reached as `self.0`. Wrapping the number in its own type means the only things you can do with it are `next`, `chance` and `below`.

The seed is saved in the project (`Band.solo: Option<u64>`, `None` with no soloist, 1 when you tick it), it is part of the band's fingerprint, and **Another solo** adds one to it. A new fingerprint means the band plays again, so a new seed plays a new solo, and undo, which brings the old seed back, brings the old solo back.

Improvising

`improvise` cuts the chorus into two-bar phrases with `chunks(2)`, and each phrase into **pulses**: a pulse is a beat, or a quarter note of a longer beat (a half note in 2/2 has two), so the soloist moves in eighth notes whatever the time signature. The line is a struct with the state a player keeps in their head: which way it is going and the last note played.


```
struct Line {
    soloist: Soloist,
    up: bool,
    last: Option<u8>,
    notes: Vec<NoteEvent>,
}
```

`phrase` first picks a chord tone for every pulse, and only then fills between them, because what goes between a beat and the next depends on where the next one lands. It starts by deciding how much of the phrase to play:

```
let late = usize::from(count >= 4 && dice.chance(35));
let rest = match count {
    6.. => 1 + dice.below(2),
    3.. => 1,
    _ => 0,
};
let Some(window) = pulses.get(late..count - rest) else {
    return;
};
```

`usize::from(bool)` turns `true` into 1 and `false` into 0. `6..` and `3..` are **range patterns**: 6 or more, then 3 or more (the arms are tried in order). `let ... else` unpacks an `Option` or leaves: `get` with a range gives a slice of the pulses, or `None` if the range does not fit, and then the phrase is not played.

The pickup uses a **let chain**: several conditions and a `let` joined with `&&` in one `if`.

```
if index == 0
    && pickup
    && pulse.start >= 2
    && let Some(over) = over_at(overs, pulse.start - 2)
{
    let approach = approach(target, over.scale, dice);
    self.play(approach, pulse.start - 2, 2, BETWEEN);
}
```

It reads as a sentence: if this is the first beat, and a pickup was rolled, and there is room before it, and a harmony sounds there, play an approach note an eighth before the beat. Before the 2024 edition of Rust, the `let` needed an `if` of its own inside the first one.

`fill` chooses what goes between two chord tones: passing notes when the scale notes between them fit in the time and each is a step from the next, a neighbour note when the two chord tones are the same, the half step between them when they are a

whole step apart, and, for a leap too wide to walk, the scale notes just before the target or a half step below it.

Joining the band

The soloist is one more `Player`, after the piano, bass, drums, guitar, strings and organ. A style never lists it among its players (`Player::Soloist | Player::Horns => false` in `BandStyle::players`); the band adds it when the seed is there:

```
if self.has_soloist() {  
  players.push(Player::Soloist);  
}
```

Everything the bigger band gives each player comes with it: its own channel with its General MIDI program, its track and pattern, being frozen when edited, **You play** to mute it and play the solo yourself. Two things are different. The soloist's notes go through the library's **melody** phrasing instead of the accompaniment's, and the shots, holds and rests do not touch it.

A song with a soloist needs at least format 19. Without one, the new fields are left out and the file is as it would be without them.

Testing it

In `harmony/chord_scale.rs`, one test per rule: `chords_of_the_key_play_its_modes`, `the_dominant_of_a_minor_key_plays_harmonic_minor`, `borrowed_chords_play_the_parallel_key`, `secondary_dominants_play_the_scale_of_their_target` and `other_chords_play_the_scale_of_their_kind`, each checking the spelled notes of a scale, as "D E F G A B C"; and `every_chord_scale_holds_its_chord`, described above.

In `band/solo.rs`, `roles_name_what_each_note_does` reads short lines by hand and checks every role, and `a_role_is_described_with_its_chord_and_scale` the sentence the piano roll shows. Three tests play many solos, over several progressions and seeds: `chord_tones_on_the_beats_and_nothing_unexplained_between` checks the rule of chord tones on the beats and that no note is left unexplained, `a_solo_is_one_line_in_range_that_breathes` that the notes never overlap, stay within a whole step of the soloist's range, are silent where there is no chord and leave a rest of an eighth or more in every two bars, and `the_same_seed_plays_the_same_solo` the dice.

In `band.rs`, `a_soloist_improvises_when_asked`,
`the_soloist_plays_the_chorus_only`,
`over_a_set_the_soloist_plays_its_notes_on_the_beats` and
`a_soloist_is_saved_in_format_nineteen` check the soloist as a member of the
band.

Your own styles: the style editor

Files: `crates/lineup-app/src/style_editor.rs`, `crates/lineup-app/src/app/styles.rs`, `crates/lineup-core/src/library/user_style.rs`, `crates/lineup-core/src/library/apply.rs` and `crates/lineup-core/src/band.rs`

The **library** is knowledge written as data: a style is a chart, a comping pattern, a bass line and a rhythm, each a few lines in a table. The **style editor** lets you write a groove those tables do not have, with the channel rack and piano roll you already know, and save it as a style of your own. The band then plays it over any chord sheet, moving every note to the chord of the moment.

Trying it

1. Open **File > New**. Press **F6** for the channel rack and click **Edit style...**. The editor opens with a copy of the band the song plays, **Pop beat copy**, in a window titled **Style editor · channel rack**.
2. Open **Start from...**, type `bossa` in **Search presets**, and choose a style. Click **Preview ×4**.
3. Click a few kick and snare steps. Click a piano note preview to open the editor's piano roll; draw a chordal figure, then press **Space** to hear it.
4. Click **Save as...**, name it `My groove`, and save it in the folder the dialog opens. Close the editor, open the **Library**, and find **My groove** among the presets (under **User styles** when they are grouped by genre).
5. Apply it with **Drums** and **Chord and bass parts** ticked and **Chord progression** unticked. The song keeps its chords and the band plays your pattern over them. Save the project, close the app, move the `.lineup-style` file away and open the project again: the band still plays it.

Where it is

- In the **channel rack**, **Edit style...** opens the style the song is playing ("Edit a copy of the current style's repeating parts", says its hint); **New style...** starts an empty one.
- **View > Style editor...** opens the editor, or brings it to the front.
- The **Library** has a **New style...** button over its list. Right-click a style or a rhythm there: a built-in one offers **Edit a copy in channel rack...**, one of yours **Edit in**

channel rack....

Edit style... starts from what the song plays. A style of yours opens as itself, so **Save style** writes back to its file. A band opens as a copy of all its players, two bars of what they play over a C chord, named "... copy". A style or rhythm being tried in a pattern opens as a copy of it, and with nothing to copy the editor starts empty: a kick, a snare, a hi-hat and a piano, over the progression **I | V | vi | IV**.

What it edits

A user style is **one repeating pattern**, one to sixteen bars long and at most 256 steps (so in 12/8, with 24 steps a bar, ten bars at most). It holds drum lanes, one for each drum sound, and at most one lane for each pitched player of the band: piano, bass, guitar, strings, organ, soloist and horns.

The top row has **Start from...**, **Save style**, **Save as...**, **Undo**, **Redo**, **Preview ×4** and **Stop preview**. Under it are **Name**, **BPM**, the time signature, **Bars** and **Swing**; then **Progression**, the chords the style brings, in Roman numerals or chord names with bars separated by **|**; **Description**; and **Part assignments**, which says which player each row is, so a row named "Rhodes" can be the piano.

The rack is the same widget as the song's, holding the draft's rows: click steps to write drums. Click a note preview to open the piano roll, where drawing, moving, stretching and the Tools menu work as in a pattern. The editor's roll has its own window and keys, and never edits the song's roll.

Preview ×4 plays the pattern four times, as written over C, and stops; **Space** starts or stops it while the editor has the keyboard, and **Stop preview** ends it early. The song pauses while it plays. **Ctrl+S** saves and **Ctrl+Shift+S** saves as.

The editor has a history of its own. **Ctrl+Z**, **Ctrl+Shift+Z** and **Ctrl+Y** undo and redo the draft without touching the song's history. Nothing reaches the song until you apply the saved style, and applying it is then one step to undo.

The music: one part over every chord

The hint under the fields says it: "Write pitched parts over C major. They follow the song's chords when applied." A player who learns a comping figure learns it as a shape, root, third, fifth, seventh, and plays the same shape over every chord. Writing

the shape over C major gives it fixed names: C is the root, E the third, G the fifth, B the seventh.

What each written note plays

Written over C	Plays over the chord
C	its root
E	its third, or the note a suspended chord has instead (F in Csus4)
G	its fifth, diminished or augmented if the chord's is
B	its own seventh (the sixth of a 6 chord); with none, the key's seventh
D, F, A	the key's note one, three or five steps above the root: the 9th, 11th or 13th
C#, Eb, F#, Ab, Bb	the same distance from the root: a colour kept

The written arpeggio C–E–G–B, in a song in C major:

Chord	Plays
Dm7	D F A C
G7	G B D F
Cmaj7	C E G B
Am	A C E G
F	F A C E
Csus4	C F G B
Am6	A C E F#

Over Am and F, which have no seventh, the B takes the key's own seventh above the root: Am plays as Am7 and F as Fmaj7, as a pianist comping in C major would colour them. If you want plain triads, leave the B out.

A scale becomes the mode of each chord

Write the whole scale, C D E F G A B, and the band plays it from each chord's root with the notes of the key:

Chord	Plays	Which is
Dm7	D E F G A B C	D Dorian
G7	G A B C D E F	G Mixolydian
Am	A B C D E F G	A Aeolian
Bdim	B C D E F G A	B Locrian
F	F G A B C D E	F Lydian

These are the modes of C major (see [Scales, modes and keys](#)): the notes of the key, started from each chord. So the D you write over C, a ninth, is the key's E over Dm7 but F over Em, and the A, a sixth, is C over Em, not C#: the line never leaves the key while the chords stay in it.

Chords from outside the key

A chord with a note outside the key, as a borrowed Fm or the E7 that leads to Am, cannot take the key's notes for its ninth or sixth. It keeps the distances written over C, the notes of its own major scale, except that a minor chord takes the flat seventh:

Chord	C D E F G A B plays	C-E-G-B plays
Fm	F G A $\flat$ B $\flat$ C D E $\flat$	F A $\flat$ C E $\flat$
E7	E F $\sharp$ G $\sharp$ A B C $\sharp$ D	E G $\sharp$ B D
B $\flat$	B $\flat$ C D E $\flat$ F G A	B $\flat$ D F A

This is simpler than the [soloist's chord scales](#), which would give the E7 that leads to Am the F and C of A minor. A style is a pattern that should sound the same over every chord; the soloist chooses notes one by one.

The chromatic notes keep their distance from the root. A written E $\flat$, the minor third over C, plays B $\flat$ over G7, the "blue" third against the chord's B; a written B $\flat$ plays C over Dm7 and E $\flat$ over F.

The bass, the register and held notes

The bass follows the same rules, with one more: its Cs play the bass note of a slash chord. A line written C-G-C-E (root, fifth, octave, third) plays D-A-D-F over Dm7, and E-G-E-E over C/E: the Cs become E, the G stays the chord's fifth and the E its third.

A part moves at most a tritone from where you wrote it. Over a chord on C to F $\sharp$ it moves up; over G to B it moves down, so a figure written around middle C stays

around it: over G7 the arpeggio is G3 B3 D4 F4, below middle C, not a fifth above.

A note held across a chord change is cut there and goes on in the new chord. Drums are copied as written. A style of two bars or more counts its bars from the start of the song: a two-bar style plays its first bar in bars 1, 3, 5 and so on, and in a bar shorter than the pattern's the notes that do not fit are left out.

Who plays what

A style with only drums, such as a copy of a rhythm, changes only the drums: the band's piano, bass and the others go on as they were. Once a style has a pitched lane, it takes over the band's harmony: the piano, bass, guitar, strings and organ play only the lanes you wrote, and one with no lane is silent. The soloist and the horns play their own parts unless the style has a lane for them. Your parts also stop for the [shots](#), [holds](#), [rests](#) and [pushes](#) of the sheet, and play louder in the [b variation](#).

Your styles are files

Save style writes to the file the style came from. **Save as...** asks for a new one. Both write `.lineup-style` files: if you type another extension, it is replaced, so `my groove.txt` becomes `my groove.lineup-style`.

The file is the style itself, written as RON (Rusty Object Notation, the format of the project files) by `serde`:

```
pub struct StyleLane {
    pub player: Player,
    pub channel: Channel,
    pub notes: Vec<NoteEvent>,
}
```



```
pub struct UserStyle {
    pub format: u32,
    pub name: String,
    pub about: String,
    pub time: TimeSignature,
    pub tempo: Tempo,
    pub swing: u8,
    pub mode: ScaleKind,
    pub chart: String,
    pub bars: usize,
    pub lanes: Vec<StyleLane>,
}
```

Each lane carries its whole `Channel` : the instrument, its General MIDI program, its volume and its other settings. Names are written in letters even when the app shows solfège, so a style saved in one language opens in the other.

Styles live in a **styles folder**, by default a `styles` folder in the app's data folder. **Edit > Settings... > Library > Your styles > Styles folder...** chooses another. The Library lists only that folder: a style saved elsewhere is not listed, and the notice after saving says so: "Saved style: ... The Library lists the styles in ...; Settings > Styles folder changes it."

The Library reads every `.lineup-style` file in the folder at startup, and again after a save or a change of folder, sorts them by name, and reports any it cannot read: "Some styles could not be loaded: ...". They join the presets beside the built-in styles and rhythms. They have no genre, so **Group by genre** puts them under **User styles (N)**. The **With chords** filter keeps those with a progression, and the instrument menu lists the sounds of their lanes. **Start from...** in the editor shows the same list and search, with **Empty style** first.

Applying one of your styles works as for any preset (see [Which parts to take](#)): **Drums** brings its drum lanes, **Chord and bass parts** its other lanes, **Voices / instruments** their sounds, and **Time signature, Tempo, Swing** and **Chord progression** those. Leave the progression unticked to keep the song's chords. Only the chord and bass parts need a band to play them, so on a song without a band, a style of your own needs them ticked for its lanes to play; its tempo, meter, swing or chords alone change the song as it is.

The style travels with the song

When **Chord and bass parts** is applied, the whole `UserStyle` is stored in the band, and when **Drums** is, in its `drums`, as `BandDrums::Saved` :

```
pub custom_rhythm: Option<Box<crate::library::UserStyle>>,
pub custom_instruments: Option<Box<crate::library::UserStyle>>,
```

The first holds the patterns, the second the instruments and their settings. The project needs no access to the `.lineup-style` file: delete it, move it, or open the song on another computer, and the band still plays. A project that carries a style of yours needs at least format 22 (see [the project file format](#)). That is also why a style can disappear from the Library without breaking a song that uses it: the song keeps its own copy.

How it works

A pattern project of its own

The editor has no second rack or second piano roll. It keeps a throwaway one-pattern `Project`, built by `UserStyle::pattern_project`, beside the style and the player of each row:

```
struct Draft {
    style: UserStyle,
    project: Project,
    roles: Vec<Player>,
}
```

It then calls the same widgets as the song windows (see [Drawing returns actions](#)). The rack returns `RackAction`s, the roll returns `RollAction`s, and the editor applies them to `draft.project` instead of the song. Adding a drum, moving a note or opening the roll go through the code that already does it. The one action it refuses is a new pattern: “Styles contain a single pattern. Save new patterns from the song piano roll.”

Its undo is the app’s own `History<T>` (see [History<T>](#)), with the whole draft as `T`:

```
history: History<Draft>,
```

From the rows back to the style

After each frame, `sync` copies the draft back into the style: meter, tempo and swing, then one `StyleLane` per channel with that channel’s player and notes, cut at the end of the pattern:

```

draft.style.time = draft.project.time_signature;
draft.style.tempo = draft.project.tempo;
draft.style.swing = draft.project.swing;

```

`UserStyle::validate` checks the result before it is saved or applied, one rule at a time, each with a message for the user:

```

if !(1..=16).contains(&self.bars) || self.length() > MAX_STEPS {
  return Err(trf("The pattern must fit in 1-16 bars and 256
steps").to_string());
}

```

```

if lane.player != Player::Drums && !players.insert(lane.player) {
  return Err(trf(
    "Assign only one lane to {}",
    &[&tr(lane.player.name())],
  ));
}

```

The others check the format, the name, the progression (every bar must read as chords), the number of lanes, the swing, each lane's settings, and that every note is inside the repeating pattern. Reading a file calls `validate` too, so a damaged or hand-edited style is refused with the reason instead of playing nonsense.

Applying: on a copy first

`Project::apply_user_style` changes the song in several steps: the band, the meter, the tempo, the chords. Reading the progression into the song's bars can fail halfway, in a meter the chart does not fit. So it works on a clone, and only a finished result replaces the song:

```

style.validate()?;
// The chart is read into the song's bars, which can fail halfway
// in another meter, after the band and the tempo changed: so it
// is applied to a copy, and an error changes nothing.
let mut project = self.clone();
project.apply_valid_user_style(style, parts)?;
*self = project;
Ok(())

```

`?` returns the error at once, and the song is left as it was. If the song has no band, one is made for the style's lanes to play in; the players the style has no lane for play as the band usually does in that time signature.

Following the chords

When the band writes its parts, `UserStyle::play_lane` repeats the style's bars over the song's and gives each pitched note to `adapt_pitch`, with the harmony and the key at that point. Its comment is the table above in words:

```
/// The note a template note written over C major plays over `harmony`,  
/// in `key`. C, E and G are the chord's root, third and fifth, and B is  
/// its seventh, or sixth, if it has one. Over a chord of the key, B, D, F  
/// and A take the note of the key as many steps above the root: an A over  
/// Em in C major is C, not C#, and a B over G is F. Over a chord from  
/// outside the key they keep their distance from the root, as in its own  
/// major, but a minor chord takes the flat seventh. The other notes are  
/// chromatic colours and keep their distance from the root.
```

The chord tones are one `match` on the written note's pitch class (0 is C, 4 E, 7 G, 11 B) and the chord's own seventh, if any:

```
let own_seventh = [10, 11, 9].into_iter().find(|&seventh| has(seventh));  
let pc = source.midi() % 12;  
let target = match (pc, own_seventh) {  
    (0, _) => 0,  
    (4, _) => intervals.get(1).copied().unwrap_or(4),  
    (7, _) => intervals.get(2).copied().unwrap_or(7),  
    (11, Some(seventh)) => seventh,  
    (11, None) if has(3) => 10,  
    _ => pc,  
};
```

`intervals` are the chord's notes in semitones above its root, so `intervals.get(1)` is its second note, the third or a suspended note, and `intervals.get(2)` its fifth.

`own_seventh` looks for a minor seventh (10), a major seventh (11) or a sixth (9), the note a 6 chord or a diminished seventh has in that place. Then D, F, A and a B with no seventh to take are counted in steps instead:

```
let steps = match (pc, own_seventh) {  
    (2, _) => 1,  
    (5, _) => 3,  
    (9, _) => 5,  
    (11, None) => 6,  
    _ => return Some(note),  
};
```

If the key has seven notes and holds every note of the chord, the note moves to the key's note that many steps above the root. Otherwise it stays as the first `match` placed it.

Saving without losing the old file

Saving writes a temporary file next to the real one and renames it:

```
let temporary = path.with_extension(format!("{EXTENSION}.tmp"));
```

A rename replaces a file in one step, so a crash or a full disk while writing leaves the old style whole, never half a file.

Testing it

In `library/user_style.rs`, `template_notes_stay_in_the_key` checks the table above note by note: a B over Am is G and over D7 C, an A over Em is C, a B over Fm is Eb and over Bb A. `every_builtin_can_be_copied_edited_and_saved` and `every_band_style_can_be_copied_with_all_its_players` copy every built-in style and band, validate the copy and read it back from its text.

`a_copy_is_named_and_described_in_the_apps_language` checks “Bossa nova copy” and “Bossa nova (copia)”, and

`user_parts_follow_chords_and_survive_a_project_round_trip` applies a two-note piano part over Dm and G7, checks the thirds it plays, saves the project (format 22), opens it again and changes the chords.

In the app, `style_window_previews_and_saves_the_editable_pattern`,

`discarding_a_draft_leaves_nothing_of_it_to_undo` and

`style_confirmations_block_draft_shortcuts_and_escape_keeps_edits` drive the editor’s window;

`saved_styles_reload_alphabetically_with_their_instruments_and_patterns` reads a styles folder back; `edit_style_copies_the_whole_band_of_a_new_song` checks that

Edit style... on a new song brings piano, bass, drums and guitar; and

`saving_a_style_elsewhere_keeps_the_styles_folder` the notice.

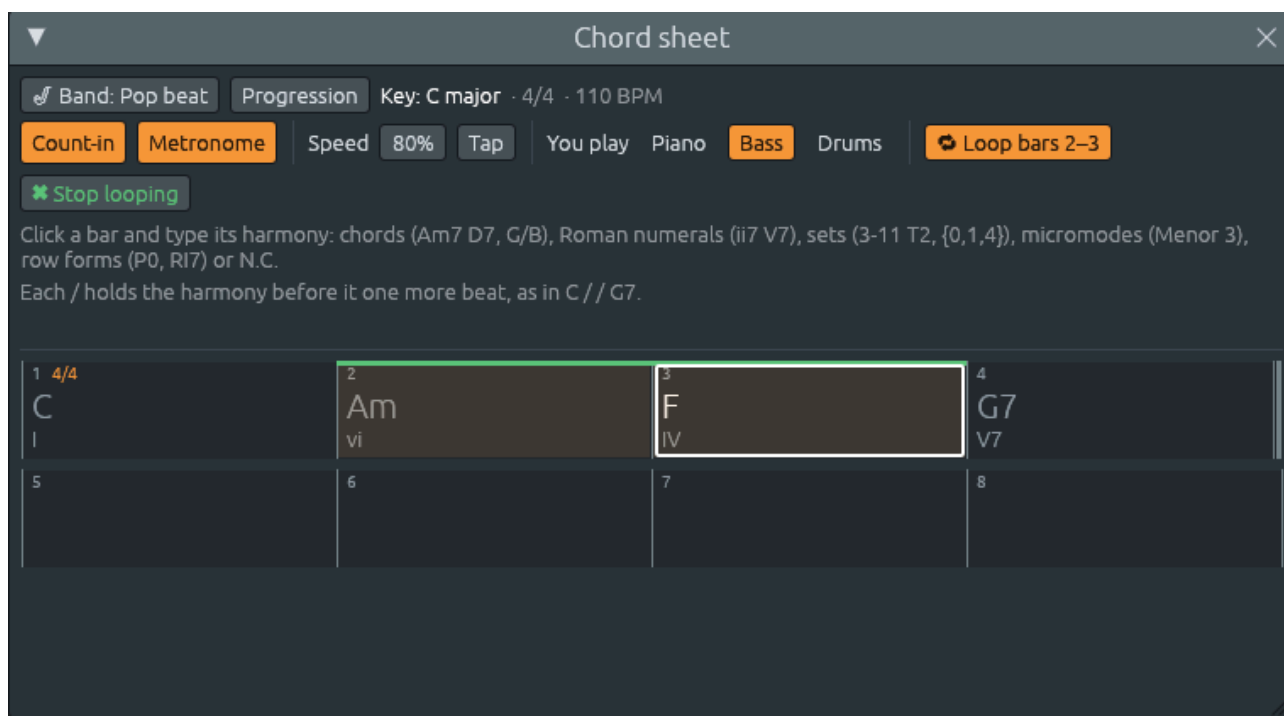
Practice tools: metronome, speed trainer, ear training

Files: `crates/lineup-audio/src/metronome.rs`, `crates/lineup-audio/src/engine.rs`, `crates/lineup-audio/src/phrase.rs`, `crates/lineup-core/src/meter.rs`, `crates/lineup-core/src/sequencer.rs`, `crates/lineup-core/src/band.rs`, `crates/lineup-core/src/history.rs`, `crates/lineup-core/src/ear_training.rs`, `crates/lineup-app/src/app/practice.rs`, `crates/lineup-app/src/chord_sheet/practice.rs`, `crates/lineup-app/src/ear_training.rs` and `crates/lineup-app/src/app/ear.rs`

A band that plays your chord sheet is most useful when you play along. This chapter is about the tools a musician practising with a recording reaches for: a **count-in**, a **metronome**, a slower **speed** and a **speed trainer** that brings it back up, **tap tempo**, taking one part away from the band to play it **yourself**, and **looping** the bars you are working on. Then the other half of practising: **ear training**, learning to name what you hear without looking.

None of this is part of the song. A saved song and a WAV export keep the song's own tempo and have no clicks in them.

Practising along



Trying it

Open the chord sheet (**F5**). Under the Band and Progression buttons is the practice row:

- **Count-in:** when you press Play, one bar of clicks comes first, and the music starts on the next “one”. Start in the middle of a bar and the count runs on to where the playhead is: from beat 3 of a 4/4 bar you hear “1 2 3 4 1 2”, and the music comes in on 3, so the beat never jumps.
- **Metronome:** a click on every beat while the music plays. The first beat of each bar is higher (A6, 1760 Hz) than the others (E6, a fourth below, 1318.5 Hz), so you always know where “one” is.
- **Speed:** how fast the band plays, as a percentage of the song’s tempo, from 25% to 200%. The song keeps its own tempo: the TEMPO box still says 110, with “TEMPO 80%” over it. **Speed trainer** beside it raises the speed a little each time the loop comes round (see [The speed trainer](#) below).
- **Tap:** click it on the beat a few times and the song’s tempo becomes yours. Taps in a row are one undo step.
- **You play:** one button for each player of the band’s style. The band leaves that part out and you play it. For every player but the drums, your keyboard (MIDI, or the typing keyboard) switches to that part’s sound.
- **Loop bars:** click a bar, Shift+click another (or hold Shift and press an arrow), and  **Loop bars 2-3** plays them over and over. **✖ Stop looping** plays the whole

song again from where you are.

Count-in and Metronome are also two icon buttons beside Play in the toolbar, so they work without the chord sheet, over any song or pattern, and **Edit > Settings... > Playback > Practice** has them with the practice speed and the speed trainer. The count-in, the metronome and the trainer's numbers are remembered for next time; the speed starts at 100% every time the app starts, with the trainer off, so a song is never slow by surprise.

Where the clicks come from

The clicks are not a channel of the song. If they were, they would be saved with it, exported into every WAV file, muted by the mixer, and moved by the band when it rewrites its parts. They are made by the audio thread itself, which already knows when every step starts.

What it needs to know is *what to click on each step*. That depends on the time signature, which can change in the middle of the song, so the UI thread works it out and sends the answer as a list, one entry per step:

```
pub enum Click {  
    #[default]  
    None,  
    Beat,  
    Bar,  
}
```

```
pub fn click(self, step: usize) -> Click {  
    if self.is_bar_start(step) {  
        Click::Bar  
    } else if self.is_beat_start(step) {  
        Click::Beat  
    } else {  
        Click::None  
    }  
}
```

“Beat” means the *felt* beat, the one you tap your foot to (see [Beats of different lengths](#)):

Time	Clicks in a bar	Why
4/4	1 2 3 4	four quarter notes
3/4	1 2 3	a waltz

Time	Clicks in a bar	Why
6/8	1 2	two dotted quarters: six eighths felt as two groups of three
2/2	1 2	two half notes, “cut time”
2+2+3/8	1 2 3	three beats, the last one longer

So a metronome in 6/8 does not click six times a bar. A jig, or Silent Night, is felt in two, and clicking every eighth note would hide that.

The song’s list comes from `Project::played_clicks`, which counts the bars as the song is *played*: a section in 3/4 that repeats is in 3/4 both times round (see [The song as it plays](#)). The selected pattern’s list comes from `Project::pattern_clicks`. They go to the audio thread as `Command::SetSongClicks` and `Command::SetPatternClicks`. When a new list arrives, the audio thread swaps it in with `std::mem::replace` and sends the old `Vec` back to the UI thread as `Garbage::Clicks`, to be freed there (see [Freeing memory somewhere else](#)).

The click itself

`ClickVoice` in `metronome.rs` plays a short sine wave that dies away:

```
const RING: f32 = 0.03;
```

```
decay: (-6.9 / (RING * sample_rate)).exp(),
```

Each frame the level is multiplied by `decay`. After `RING` seconds (30 ms, 1440 frames at 48 kHz) it has been multiplied by `decay` 1440 times, which is $e^{-6.9} \approx 0.001$, a thousandth of where it started: -60 dB, too quiet to hear. Long enough to hear a pitch, short enough to be over well before the next beat even at 300 BPM.

The count-in: a second clock

The audio thread has one `Sequencer`, the clock that says when each step starts (see [The sequencer](#)). It also drives the playhead, which the window reads to draw where the song is. During a count-in the song has not started, so the playhead must not move. The count-in gets a clock of its own, `counter`, and a small record of how far it has got:

```

/// A count-in under way: the bar it counts (where it starts in the
/// loop's clicks, and how many steps it has), how many steps it lasts in
/// all, and how many have gone by.
#[derive(Debug, Clone, Copy)]
struct CountIn {
    bar: usize,
    length: usize,
    steps: usize,
    counted: usize,
}

```

Play starts it:

```

let (bar, length) = count_in_bar(self.loop_clicks(), self.start);
let length = length.max(1);
let into_the_bar = self.start.saturating_sub(bar) % length;
self.counter.set_tempo(self.tempo_of_step(self.start));
self.counter.reset();
self.counting = Some(CountIn {
    bar,
    length,
    steps: length + into_the_bar,
    counted: 0,
});

```

`count_in_bar` looks back from the playhead to the last `Click::Bar`, and on to the next, to find the bar the playhead is in. The count lasts that whole bar *plus* the steps from its start to the playhead: from step 6 of a 16-step bar, $16 + 6 = 22$ steps, “1 2 3 4 1 2”, and the music starts on step 6, half a beat into beat 2. It counts at the tempo where the music will start, so a song that begins slowly is counted in slowly.

Then, on every frame:

```

if let Some(count) = self.counting {
    self.count_in_frame(count);
}

```

```

fn count_in_frame(&mut self, mut count: CountIn) {
    if self.counter.tick().is_none() {
        return;
    }
    if count.counted == count.steps {
        self.counting = None;
        self.sequencer.start_at(self.start);
        self.playing = true;
        self.trace_play(self.start);
        return;
    }
    let step = count.bar + count.counted % count.length;
    self.click.strike(click_at(self.loop_clicks(), step));
    count.counted += 1;
    self.counting = Some(count);
}

```

When the count is over, the song's own sequencer starts *on that same frame*: the frame loop handles the count-in before it asks the song's sequencer for a step, so the first note lands exactly one bar after the first click. The test

`the_count_in_comes_before_the_music` checks it to the frame: at 120 BPM a step is 6,000 frames at 48 kHz, the clicks come at frames 0, 24,000, 48,000 and 72,000 (each heard one frame in, as a sine starts at zero), the playhead waits, and the first note sounds at 96,000.

`CountIn` derives `Copy`, so if `let Some(count) = self.counting` makes a copy, and `count_in_frame` gets the copy. If it took a reference into `self.counting` instead, that borrow of `self` would still be alive while the function calls `self.counter.tick()` and `self.click.strike(...)`, which need `&mut self`: the borrow checker would refuse. Working on a copy and writing it back at the end (`self.counting = Some(count)`) is the usual way out for small state like this.

Stop, during the count-in, ends it. Moving the playhead or switching between pattern and song mode during the count starts it again, counting the bar playback will now start in. Keys played during the count-in are not recorded: the song has no position until the music starts, and recording keeps only notes that have one.

The speed

The practice speed never touches the project. `Project::tempo` and the tempo markers stay as they are; only what the audio thread is told changes:

```
pub(super) fn at_speed(tempo: Tempo, speed: f32) -> Tempo {
    if speed == 1.0 {
        tempo
    } else {
        Tempo::new(tempo.bpm() * speed)
    }
}
```

Every command that carries a tempo goes through it: `SetTempo` after an edit, and the tempo of every step of the song, so a song that slows down at the end still slows down, from 80% of its tempo. A WAV export builds its commands at speed `1.0`, which is why it is never slowed.

Tap tempo

Tap tempo is arithmetic on the times of the taps. Taps at 10.0, 10.5, 11.0 and 11.5 seconds are three beats in 1.5 seconds:

$$\text{BPM} = 60 \times \text{beats} \div \text{seconds} = 60 \times 3 \div 1.5 = 120$$

Using the first and the last tap of the run, rather than only the last two, averages out the unevenness of a human hand. Only the last eight taps count, so a tempo that settles as you tap is followed, and a pause of more than two seconds starts a new run. The result is rounded to a whole BPM.

You tap the beats the metronome clicks, and they are not always quarter notes, while a tempo in BPM counts quarter notes. So the app first works out how many quarter notes a felt beat is:

```
let time = self.project.time_signature;
let beats = time.beat_starts().len().max(1);
let quarters = time.steps_per_bar() as f64 / beats as f64 /
f64::from(STEPS_PER_BEAT);
```

```
let bpm = (60.0 * quarters * beats as f64 / (last - first)).round() as
f32;
```

In 6/8 a bar is 12 steps and has two beats, so a beat is 6 steps, a dotted quarter, 1.5 quarter notes. Tapping 80 beats a minute sets 120 BPM. In 2/2 a beat is a half note, two quarters. In plain 7/8 every eighth is a beat; in 2+2+3/8 the three beats are uneven, so a tapped beat counts as their average, 7/6 of a quarter note.

Each tap is a click, and each click normally starts a new undo step (see [Several clicks, one step](#)). Undoing a tempo tap by tap would be tiresome, so `History` has one method for it:

```
pub fn reopen(&mut self, label: &'static str) -> bool {
    if self.redo.is_empty() && self.undo_label() == Some(label) {
        self.open = true;
    }
    self.open
}
```

From the third tap of a run on, the app reopens the “tap tempo” step before changing the tempo, so the whole run is one step. It only does so when the last step *is* “tap tempo” and nothing was undone since: it never joins a tap to some other change.

You play

Taking a part away from the band mutes that player’s track in the playlist:

```
pub fn set_you_play(&mut self, player: Player, you: bool) -> bool {
    let Some(track) = self.player_track(player) else {
        return false;
    };
    self.playlist.tracks[track].muted = you;
    true
}
```

Muting, not removing, matters. The band goes on writing the part when you change a chord, so when you click the button again it plays what the sheet says *now*. And because a part’s fingerprint leaves out the track its clips are on (see [A part you edit stays yours](#)), muting does not count as an edit: the part is not frozen.

For every player but the drums, the app also selects that player’s channel, the one your keyboard plays: you take over the bass *with* the bass’s sound. The drums play on several channels (kick, snare, hi-hat...), so there is no one channel to select for them.

Looping the bars you choose

The sheet remembers, besides the bar selected, an `anchor`: the other end of a selection made with Shift.

```
fn select_to(&mut self, bar: usize, shift: bool) {
    self.anchor = match (shift, self.anchor, self.selected) {
        (false, ..) => None,
        (true, Some(anchor), _) => Some(anchor),
        (true, None, selected) => selected,
    };
    self.selected = Some(bar);
    self.clipboard = None;
}
```

The first Shift+click keeps the bar selected before as the anchor; later ones move only the other end, as in a text editor. Delete clears every bar selected, in one undo step.

Loop bars turns the selection into the playlist's loop region, which song mode already plays (see [Sections, repeats and loops](#)), switches to song mode and moves to the loop's start. Bars past the end of the song cannot be looped: there is nothing there to play, and the button's hint says so. On the sheet, the bars that loop get a green line along their top, the colour of the loop on the playlist.

The speed trainer

A passage you cannot play at full speed is learned the same way on every instrument: slowly, until it is right, then a little faster, and again, until it is up to tempo. With a metronome that means stopping every few times round to turn a knob. The **speed trainer** turns the knob for you: each time the loop comes round, the band plays a few percent faster, until it reaches the speed you are aiming for.

The screenshot shows the 'Chord sheet' application window. At the top, there's a title bar with a close button. Below it, a control bar contains buttons for 'Band', 'Progression', 'Key: C major · 4/4', and 'Write the chords the song plays'. A 'Show' dropdown is set to 'As written'. Below this, there are buttons for 'Count-in', 'Metronome', 'Speed' (set to 80%), 'Tap', and 'Speed trainer' (highlighted in orange). To the right of 'Speed trainer' are sliders for 'from 70% to 100%' and 'by 5% every 1 pass'. Below the sliders are two buttons: 'Loop bars 3-4' (orange) and 'Stop looping' (green). A text area provides instructions: 'Click a bar and type its harmony: chords (Am7 D7, G/B), Roman numerals (ii7 V7), sets (3-11 T2, {0,1,4}), micromodes (Menor 3), row forms (P0, R17) or N.C. Each / holds the harmony before it one more beat, as in C // G7. The band stops with a shot (C.), a hold (C..) or a rest (C...), and ^C comes in early. Ctrl+C and Ctrl+V copy and paste bars as text.'

The main area displays a grid of 12 bars. Bars 3 and 4 are highlighted with orange borders, indicating they are the current loop region. The chords for these bars are C (I) and Am (vi) for bar 3, and F (IV) and G (V) for bar 4. The other bars follow a similar pattern: Bar 1 (C, Am), Bar 2 (F, G), Bar 5 (C, Am), Bar 6 (F, G), Bar 7 (C, Am), Bar 8 (F, G), Bar 9 (C, Am), Bar 10 (F, G), Bar 11 (C, Am), and Bar 12 (F, G).

Trying it

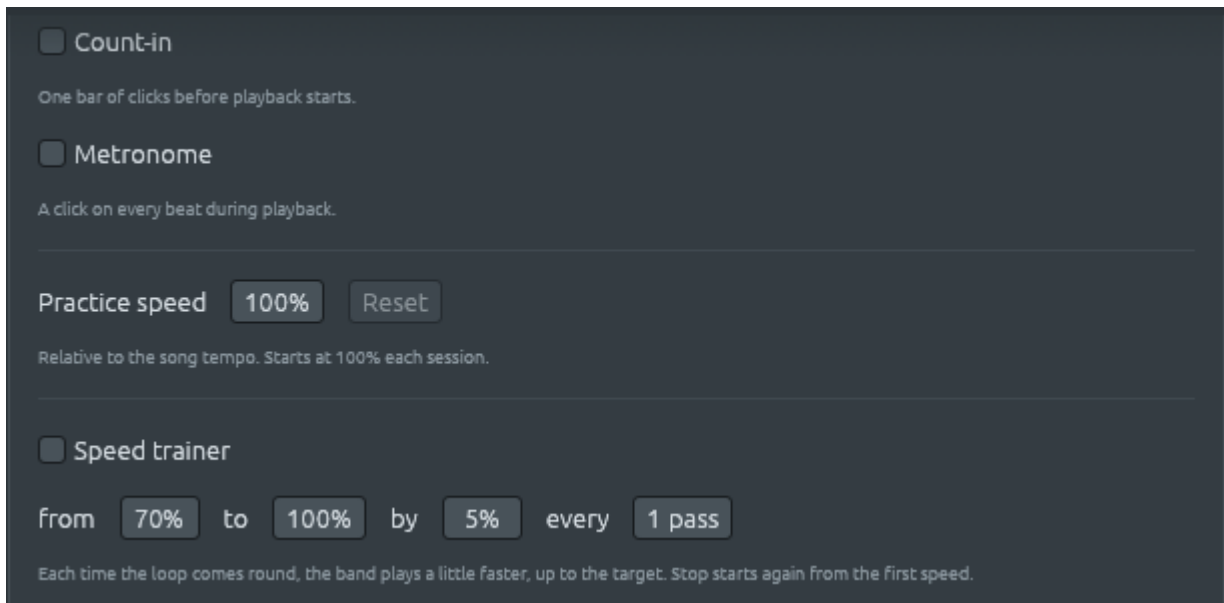
1. Press **Ctrl+N** for a new song with a band, at 110 BPM, and type four bars on the chord sheet: | C | Am | F | G | . Click bar 3, Shift+click bar 4 and click 
Loop bars 3–4.
2. Click **Speed trainer**. The speed drops to **70%**, and four boxes appear beside the button: **from 70% to 100% by 5% every 1 pass.**
3. Press Play. The band starts at 70% of 110 BPM, 77 BPM. Each time bar 4 ends and bar 3 comes round again, the speed goes up by 5% of the song's tempo: 75%, 80%, 85%... On the seventh time round it plays at 100%, the song's own tempo, and stays there. The status bar says where it is: "Speed trainer: 80%, on the way to 100%".
4. Press **Stop** (■). The speed goes back to 70%, ready to start the next run slow again. **Pause** (■) keeps the speed, so Play carries on from where you were, at the speed you had got to.

The four boxes are dragged, or double-clicked to type:

- **from:** the speed Play starts at.
- **to:** the speed to get to. 100% is the song's tempo; 110% goes past it, to have some room to spare when you play it for real. A target below the start slows down instead.
- **by:** how much faster each new speed is, from 1% to 25% of the song's tempo. The last step stops at the target, so 96% by 8% gives 100%.
- **every:** how many times the loop plays at each speed before the next one, from 1 to 16. With **every 2 passes** you play each speed twice: once to get it, once to check that you have it.

A speed you set by hand in the **Speed** box, during a run, is where the trainer carries on from. Clicking **Speed trainer** again turns it off and leaves the speed where it got to.

"The loop" is whatever playback goes round: the bars you loop, the whole song when nothing is looped (it starts again from bar 1 at its end), or the pattern, in pattern mode. The trainer works with all three. The same controls are in **Edit > Settings... > Playback**, under the practice speed:



Counting the times round

To go faster “each time the loop comes round”, the app has to know when that happens. The window could watch the playhead and call it a lap whenever the step number goes down, from the last step back to the first. But the step number also goes down when you click an earlier bar while the music plays, and that is not a time round: the trainer would speed up whenever you jumped back to try a bar again.

The one that knows is the audio thread’s `Sequencer`, which counts its **laps**: the times it played on past the last step and came round to step 0 (see [Ticking](#)). Inside `tick`:

```
if self.coming_round {
    self.laps += 1;
}
self.next_step = (self.next_step + 1) % self.length;
self.coming_round = self.next_step == 0;
```

`coming_round` is set when the last step fires, and the lap is counted when step 0 fires after it. A jump (`start_at`, which a click on a bar sends), a restart (`reset`) and a loop made shorter than where playback had got to (`set_length`) all clear `coming_round`, so none of them counts.

The audio thread stores the count after every step in an atomic integer, as it does the playhead, and `AudioEngine::laps()` reads it. The window looks at it once a frame:


```
fn came_round(&mut self, laps: usize) -> Option<u16> {
    let seen = self.laps_seen.replace(laps);
    if !self.training {
        return None;
    }
    // A new audio thread, after a restart, counts from 0 again.
    let new = laps.checked_sub(seen)?;
    if new == 0 {
        return None;
    }
    // ...
}
```

Comparing with the count it saw last, rather than looking for “a lap happened”, means a lap is never missed: if the window was hidden for a few seconds and did not draw, it finds several new laps at once and moves the speed on once for each.

In a function that returns an `Option`, `value?` means “the value inside, or return `None` from here”. `seen?` returns `None` the first time, when nothing was seen yet.

`checked_sub` is a subtraction that gives `None` instead of going below zero: after **Restart audio** a new audio thread counts from 0 again, `laps` is smaller than `seen`, and the trainer waits for the next lap instead of working out a huge number of them. A plain `laps - seen` would panic there in a debug build, and wrap round to a number near 2^{64} in a release build.

On the beat, not a step late

The window draws about 60 frames a second, so it learns of a lap up to a frame after it happened, and the audio thread then needs a block more to get a command. By then step 0 of the new time round has started, at the old tempo, and the new speed would start on step 1: the first sixteenth note of every pass a little too slow. Nobody would notice one sixteenth 5% too long, but the click on the first beat would be a few milliseconds late, every time round.

So the window does not wait for the lap. As soon as it knows the next speed (right after the last lap, or when you turn the trainer on), it sends the audio thread the tempos of that speed, to be used **from the next lap on**:

```
pub struct LapTempos {
    /// In place of the [Command::SetTempo] tempo.
    pub tempo: Tempo,
    /// In place of the [Command::SetSongTempos] list.
    pub song: Vec<Tempo>,
}
```

```
SetLapTempos(Option<LapTempos>),
```

The audio thread keeps them until the loop comes round. Just before a step fires it already asks which tempo that step is at; first it asks whether this is step 0 coming round, and if so takes the waiting tempos in:

```
if let Some(step) = self.sequencer.due() {  
    if self.sequencer.comes_round()  
        && let Some(tempos) = self.lap_tempos.take()  
    {  
        self.tempo = tempos.tempo;  
        let old = std::mem::replace(&mut self.song_tempos, tempos.song);  
        self.throw_away(Garbage::Tempos(old));  
    }  
    let tempo = self.tempo_of_step(step);  
    if tempo != self.tempo_now {  
        self.set_tempo(tempo);  
    }  
}
```

`take()` empties the slot, so the tempos are used once. The old list goes back to the UI thread to be freed, as every list the audio thread lets go of does.

When the window then sees the lap, it sends the same tempos again with the usual `SetTempo` and `SetSongTempos`. When they arrived in time that changes nothing: they are the very same numbers. It matters when they did not: if the request reached the audio thread after the loop had already come round, the resend sets the new speed a step late, and the next request replaces the old one before it can be played a lap too late.

Keeping the request up to date

A request waits a whole time round, and the song can change in the meantime: you type a chord, and the band's new part goes to the audio thread with the song's tempo list. A tempo list worked out before that could be the wrong length, or miss a tempo change you just made. So every time the window sends `SetTempo` or `SetSongTempos`, it counts it (`tempos_sent`), and it keeps, beside the speed change it asked for, the count at the time. Once a frame it compares them with what it would ask for now:

```

let change = self.practice.next_change();
let sent = change.map_or(0, |_| self.practice.tempos_sent);
if self.practice.lap_change != Some((change, sent)) {
    let tempos = change.map(|(_, next)| {
        let (tempo, song) = self.tempos_at(next);
        LapTempos { tempo, song }
    });
    self.send(Command::SetLapTempos(tempos));
    self.practice.lap_change = Some((change, sent));
}

```

`next_change` is `None` while the trainer is off, at its target, or before the last pass at a speed when it plays each speed more than once; then the window sends `SetLapTempos(None)`, which takes back any tempos still waiting. Stop, a speed typed by hand, or new trainer numbers all change `next_change`, so the request follows them too.

How long a run takes

Two bars of 4/4 are 8 beats. With the defaults over bars 3–4 of the song above, at 110 BPM:

Time round	Speed	BPM	Starts at
1	70%	77	0 s
2	75%	82.5	6.23 s
3	80%	88	12.05 s
4	85%	93.5	17.51 s
5	90%	99	22.64 s
6	95%	104.5	27.49 s
7	100%	110	32.08 s

Eight beats at 77 BPM last $8 \times 60 \div 77 = 6.23$ seconds; at 82.5 BPM the next eight last 5.82, and $6.23 + 5.82 = 12.05$. Each change comes on the loop's first step, which the [playback trace](#) shows as a `# tempo:` line at step 0.

Ear training

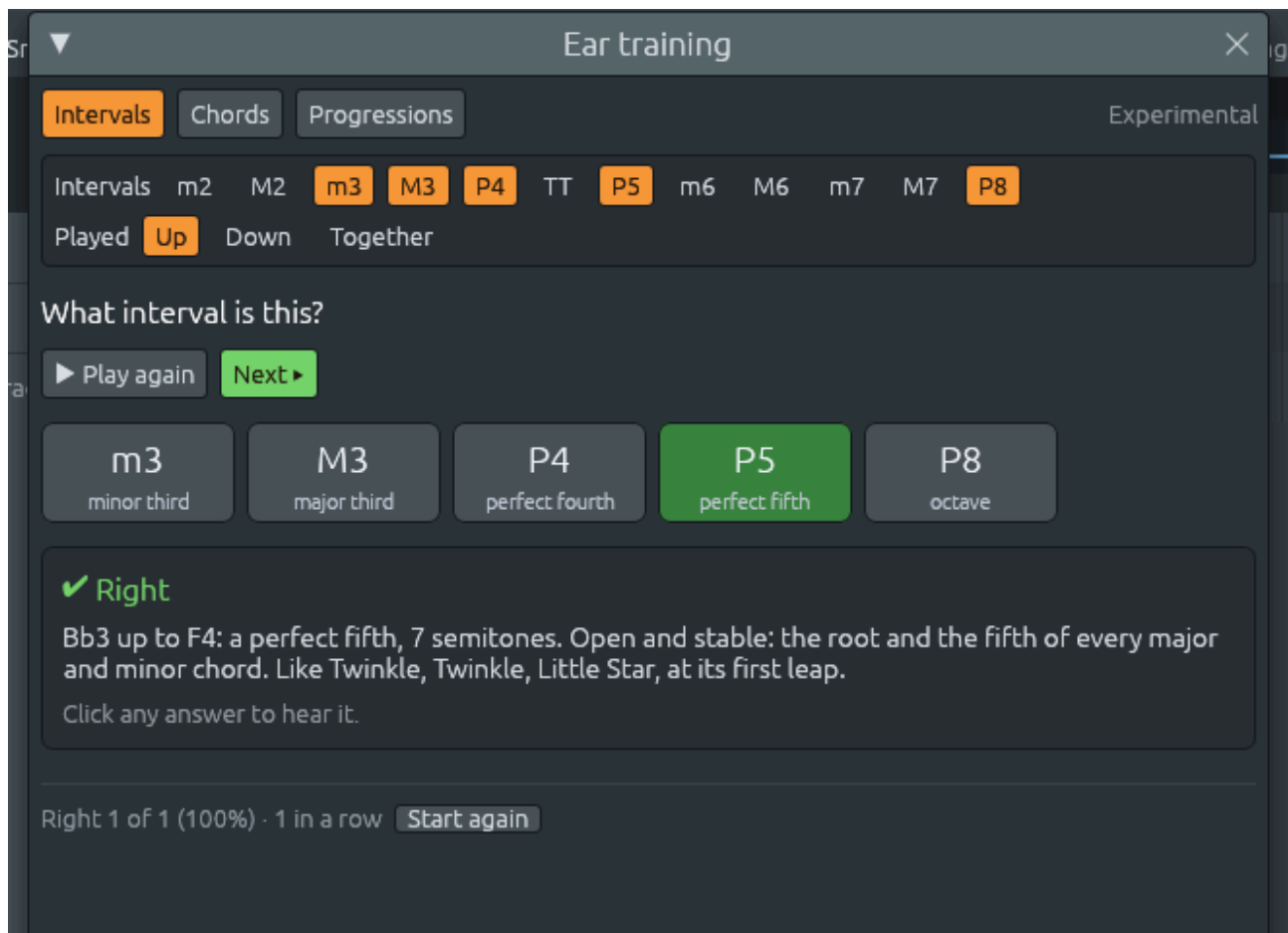
Everything else in the app helps you see harmony: chord names on the sheet, numerals and colours on the harmony keys, set names in the twelve-tone window.

Musicians also need to *hear* it: to tell a minor third from a major one, a diminished chord from a minor one, and the V of a song from its IV, without looking. That is a skill of its own, and it is learned the way ear-training games teach it: something plays, you name it, and you are told at once whether you were right.

It is **experimental** and stays out of the way: it lives in its own window, which is closed until you open it, and it changes nothing in the song or in the file.

Trying it

View > Ear training opens the window. Three tabs sit at the top: **Intervals**, **Chords** and **Progressions**. Under them, a box says what to ask; then ► **Start** asks the first question, and plays it. If the song was playing, it pauses, so the question is heard alone.

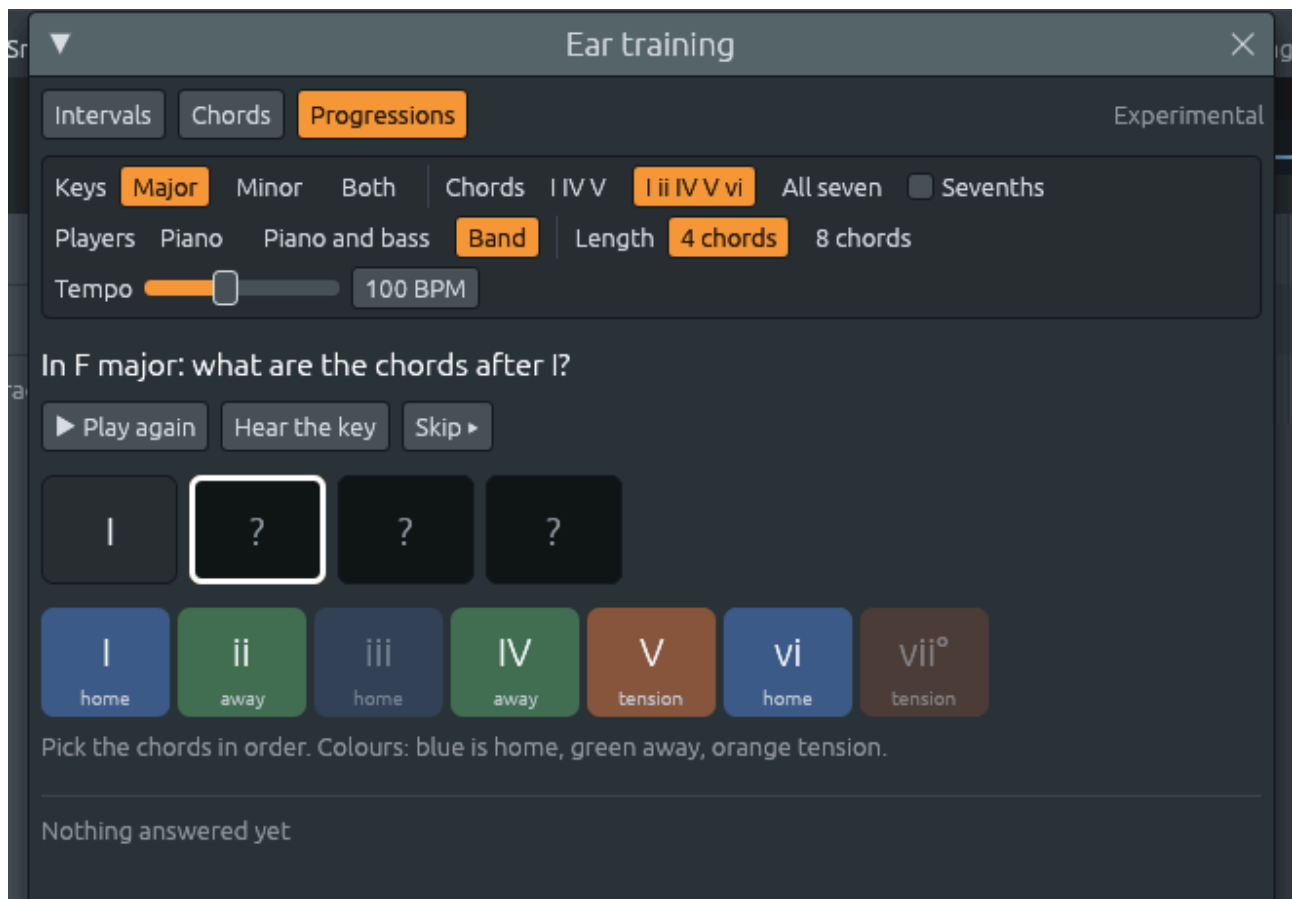


Intervals. Two notes play, and the big buttons are the answers. The box at the top picks which intervals can be asked (a minor third, a major third, a fourth, a fifth and the octave to begin with), and how they are played: **Up** (low note first), **Down** (high note first) or **Together**. Fewer choices make it easier; start with a few that sound very different, and add the others one at a time.

After you answer, your button turns green or red, the right one turns green, and a box explains the answer: the two notes with their octaves, the interval's full name, how many semitones it spans, what gives it its sound and, going up, a tune that starts with it. Then every answer button plays *its own* interval from the same note, so you can hear the one you said next to the right one. ► **Play again** plays the question as often as you like; **Next** ► asks another (it is **Skip** ► before you answer, and a skipped question does not count).

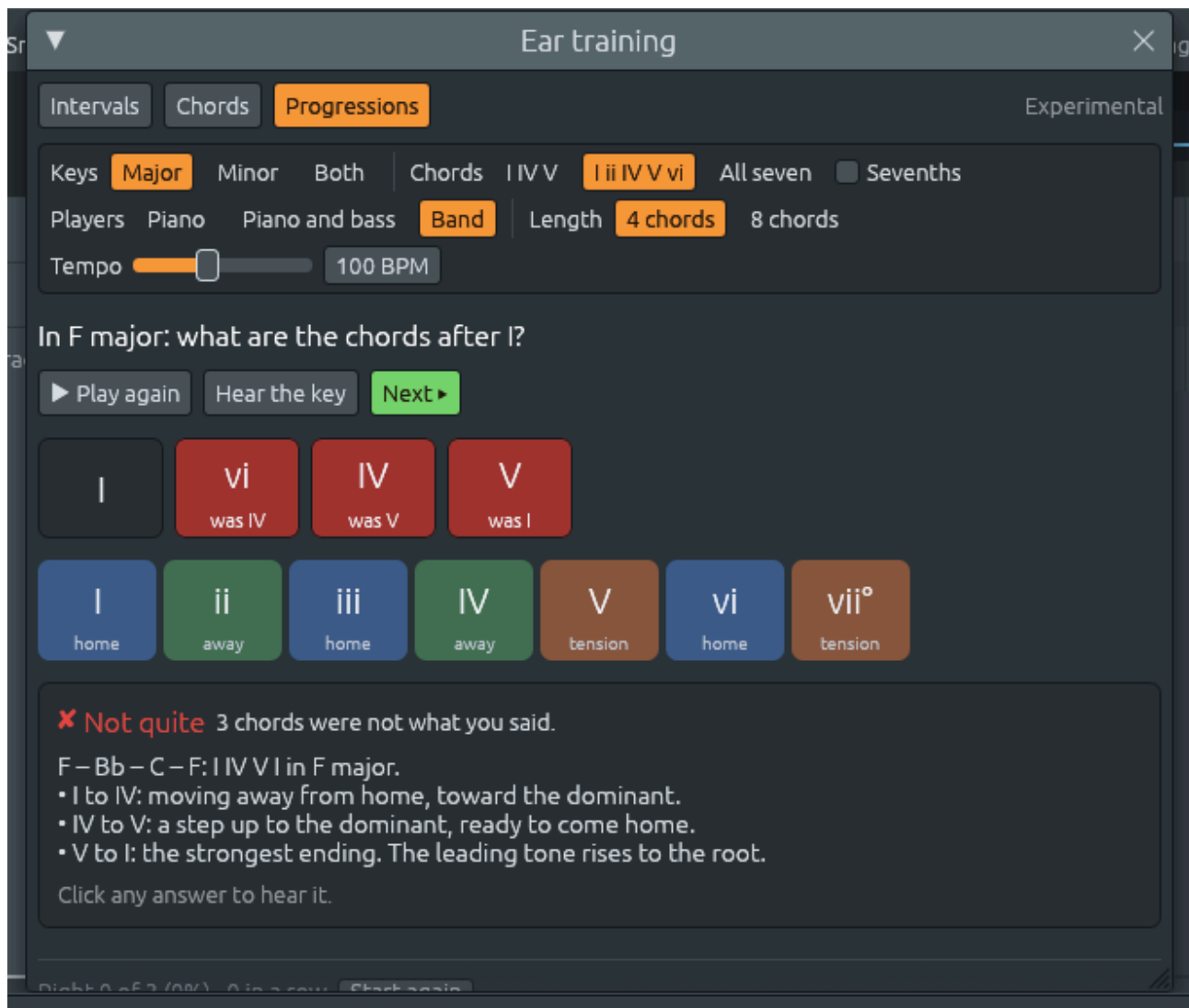
The screenshot shows the 'Ear training' application window. At the top, there are tabs for 'Intervals', 'Chords', and 'Progressions', with 'Chords' selected. Below the tabs, there are buttons for chord types: 'maj', 'm', 'dim', 'aug', '7', 'maj7', 'm7', 'm7b5', and 'dim7'. A checkbox for 'Inversions' is checked. The question is 'What kind of chord is this?'. Below the question are three buttons: 'Play again', 'Note by note', and 'Next ►'. The answer buttons are arranged in two rows: 'maj' (major), 'm' (minor), 'dim' (diminished), 'aug' (augmented), '7' (dominant seventh) in the first row, and 'maj7' (major seventh), 'm7' (minor seventh), 'm7b5' (half-diminished seventh), 'dim7' (diminished seventh) in the second row. The 'dim7' button is highlighted in green, indicating it is the correct answer. Below the answer buttons, a feedback box shows a red 'X' and the text 'Not quite You said major.' followed by an explanation: 'Ddim7: D diminished seventh, root position. D F Ab Cb: a minor third, a minor third and a minor third stacked up from D. Cb is the B key, written Cb because it is the chord's seventh: seven letters from D (D E F G A B C). Minor thirds all the way up, splitting the octave in four equal parts: the vii°7 of harmonic minor.' and a prompt 'Click any answer to hear it.' At the bottom, a progress bar shows 'Right 1 of 4 (25%) - 0 in a row' and a 'Start again' button.

Chords. A chord plays, and you name its kind: major, minor, diminished, augmented, or one of five seventh chords. The root changes from question to question; only the kind counts. **Note by note** plays the chord one note at a time from the bass up, each held, so it ends as the whole chord. With **Inversions** ticked, the lowest note is not always the root.



Progressions. The band plays a few chords in a key, one bar each, and you name them by their numerals. The question says the key ("In F major: what are the chords after I?") and the first chord is always I, home, so you know where you start. The answers are the harmony keys, with their colours and their job (home, away, tension) but with the chord names hidden. Keys for chords that cannot be asked are dim. While the band plays, a white frame walks along the slots, as it does on the harmony keys, so you know which bar you are hearing.

Click the keys in order, one per chord after I. Then each slot turns green or red, a red one says what the chord was ("was IV"), and the box spells out the chords, their numerals and why each one follows the one before, in the words of the harmony keys' hints:



After answering, every slot and every key plays its chord on the piano, and **Hear the key** plays I, IV, V and I: a cadence that says where home is, which helps when you lose your way.

The box at the top picks the **Keys** (Major, Minor or Both), the **Chords** that can come up (I IV V; I ii IV V vi; or All seven), whether they are **Sevenths** (V7, ii7, Imaj7...), the **Players** (Piano, Piano and bass, or the Band with drums), the **Length** (4 or 8 chords) and the **Tempo** (60 to 160 BPM).

The score. The line at the bottom counts right answers, the percentage and how many in a row. Each tab keeps its own, and in a progression every chord counts as one answer. **Start again** sets it back to zero. The score is forgotten when the app closes; the choices in the boxes are remembered.

The theory in the answers

The app teaches with its answers, so they have to be exactly right. This is what they say, and why.

Intervals

An interval's name has two parts. The **number** counts letters, both ends included: C up to E is C, D, E, three letters, a *third*. C up to E $\flat$ is a third too: same letters. The **quality** then comes from the semitones: a third of 4 semitones is *major*, of 3 *minor*.

Seconds, thirds, sixths and sevenths come in major and minor. Unisons, fourths, fifths and octaves are called **perfect** instead: turned upside down (the low note moved up an octave), a perfect interval gives another perfect one, while a major one becomes minor and a minor one major. The numbers of an interval and its upside-down twin add up to nine, their semitones to twelve: a major third (4) turned over is a minor sixth (8), a perfect fourth (5) a perfect fifth (7). One semitone more than major or perfect is *augmented*, one less than minor or perfect *diminished*.

Button	Name	Semitones	From C	A tune that starts with it
m2	minor second	1	C–D $\flat$	Jaws, the shark's two notes
M2	major second	2	C–D	Happy Birthday, at its first step up
m3	minor third	3	C–E $\flat$	Greensleeves
M3	major third	4	C–E	When the Saints Go Marching In
P4	perfect fourth	5	C–F	Here Comes the Bride
TT	tritone (augmented fourth)	6	C–F $\sharp$	Maria, from West Side Story
P5	perfect fifth	7	C–G	Twinkle, Twinkle, Little Star, at its first leap
m6	minor sixth	8	C–A $\flat$	The Entertainer, at its first leap up
M6	major sixth	9	C–A	My Bonnie Lies over the Ocean

Button	Name	Semitones	From C	A tune that starts with it
m7	minor seventh	10	C–B $\flat$	Somewhere, from West Side Story
M7	major seventh	11	C–B	
P8	octave	12	C–C	Over the Rainbow

The tunes start with the interval going up, so the app names one only when the question went up. The major seventh has none: few well-known tunes start with it.

The **tritone** is the one interval with two numbers. Six semitones up from C is the F $\sharp$ key or the G $\flat$ key: C to F $\sharp$ is four letters (C D E F), an *augmented fourth*; C to G $\flat$ is five (C D E F G), a *diminished fifth*. Same keys, same sound, two spellings. Its name says what it is: three whole tones, exactly half an octave.

Spelling. The explanation writes the notes as a musician would. The bottom note is a white key, or a black key written with a sharp or a flat, whichever writes the *pair* with fewer of them: C $\sharp$ up to E is a minor third (writing it D $\flat$ up to F $\flat$ would take two flats), but D $\flat$ up to F is a major third (C $\sharp$ up to E $\sharp$ would take two sharps). When both are as simple, a black key keeps its usual name: B $\flat$ up to D $\flat$, not A $\sharp$ up to C $\sharp$. The tritone is an augmented fourth or a diminished fifth by the same rule: F up to B (no accidental) rather than F up to C $\flat$, and an augmented fourth when both are as simple: C up to F $\sharp$ rather than C up to G $\flat$, B $\flat$ up to E rather than B $\flat$ up to F $\flat$ or A $\sharp$ up to E.

Chords

Every chord the exercise plays is built of thirds, stacked up from the root (see [Chords](#)). The nine kinds, on C:

Button	Name	Symbol	Notes	Thirds, from the bottom
maj	major	C	C E G	major, minor
m	minor	Cm	C E $\flat$ G	minor, major
dim	diminished	Cdim	C E $\flat$ G $\flat$	minor, minor
aug	augmented	Caug	C E G $\sharp$	major, major
7	dominant seventh	C7	C E G B $\flat$	major, minor, minor
maj7	major seventh	Cmaj7	C E G B	major, minor, major

Button	Name	Symbol	Notes	Thirds, from the bottom
m7	minor seventh	Cm7	C E $\flat$ G B $\flat$	minor, major, minor
m7b5	half-diminished seventh	Cm7b5 (C $\emptyset$ 7)	C E $\flat$ G $\flat$ B $\flat$	minor, minor, major
dim7	diminished seventh	Cdim7 (C $^{\circ}$ 7)	C E $\flat$ G $\flat$ B $\flat\flat$	minor, minor, minor

Two chords that are often taught early are **left out on purpose**: sixth chords and suspended chords. C6 is C E G A, the very same keys as Am7 (A C E G); Csus2 (C D G) has the keys of Gsus4 (G C D). Played on their own, nothing tells them apart: what makes one C6 and the other Am7 is the bass and the music around it. A quiz that asked them would mark right answers wrong. A test checks that no two of the nine kinds share their notes on any two roots.

Some spellings look odd at first: B $\flat\flat$ in Cdim7, C $\flat$ in Fdim. They are not mistakes. The seventh of a C chord is some kind of B (C D E F G A B, seven letters), and a diminished seventh is a semitone below the minor seventh B $\flat$, so it is written B double flat, though it is the A key. In the same way the fifth of F is some kind of C, and a diminished fifth is a semitone below C: C $\flat$, the B key. The explanation says so whenever it writes a note with a letter other than its key's usual names, because the letter says which job the note does in the chord.

Inversions. A chord is *inverted* when a note other than its root is lowest: the third (first inversion), the fifth (second inversion) or the seventh (third inversion). The symbol names the bass after a slash: C/E is C major with E lowest (see [Inversions](#)). An inverted chord keeps its kind, so the answer does not change, but it sounds less settled and is harder to name; that is what the **Inversions** box is for.

Two kinds are special. The augmented chord's notes are evenly spaced (a major third apart, and a major third from the top note back up to the root), and so are the diminished seventh's (a minor third apart). Put any of their notes in the bass and you get the very same keys as the chord in root position on *that* note: Caug/E is also Eaug (E G $\sharp$ B $\sharp$, where B $\sharp$ is the C key). The ear cannot tell which note is the root, so the explanation of an inverted augmented or diminished seventh chord says which other chord it also is. The kind is still certain, so the question stays fair.

Progressions

The numerals and their colours come from the harmony keys (see [The harmony keys](#)): upper case for major chords, lower case for minor, $^{\circ}$ for diminished, and three jobs,

home (blue), away (green) and tension (orange). In C major:

I	ii	iii	IV	V	vi	vii°
C	Dm	Em	F	G	Am	Bdim

A minor key reads *i ii° bIII iv V bVI bVII*. The numerals count along the *major* scale from the same tonic, so the chords on the lowered third, sixth and seventh notes get a flat: in A minor, C is bIII. And V is major, as in nearly all music in minor keys: E major in A minor, with G# pulling up to A. With **Sevenths** the keys become *I^{maj7} ii⁷ iii⁷ IV^{maj7} V⁷ vi⁷ vii^{ø7}* in major and *i⁷ ii^{ø7} bIII^{maj7} iv⁷ V⁷ bVI^{maj7} bVII⁷* in minor.

How a progression is made. It starts on I. Each next chord is usually one of the moves the harmony keys suggest after the chord before (the same list behind their faint yellow hints, see [What comes next](#)), the most usual move most often. One time in four it is any chord chosen instead, so that chords the usual moves seldom reach, such as iii, are heard too. It never plays the same chord twice in a row, comes back to I only at the end, and ends on I or V: a full stop, or a half cadence, a question waiting for its answer.

What the band plays. Each chord lasts a bar of four beats. The piano plays it on beats 1 and 3, each chord voiced close to the one before, as a pianist's hands move (see [Playing a key](#)). The bass plays the root on the same beats, in its low octave. The drums keep time: bass drum on 1 and 3, snare on 2 and 4, hi-hat on every eighth. The last chord is held for the whole bar over a crash. With the **Piano** alone, the piano also plays the root under the chord: without a bass, a close voicing could be heard as an inversion.

Listening hints.

- **Follow the bass.** It plays the roots, so its line *is* the progression. A root falling a fifth (V to I, ii to V) sounds like arriving.
- **I and V are the easiest pair.** V is tension, I is rest; V almost asks for I.
- **IV and ii both lead away and to V.** They share two notes (F A in C major: F A C and D F A), so they sound alike; ii is the darker one.
- **vi is a sad home.** It shares two notes with I (C E G and A C E), and so does iii (E G B). If a chord feels like home but minor, it is probably vi.
- **Hear the key** whenever you lose track of where home is.

How it is built

The work splits along the lines the app already has. `lineup-core` makes the questions and their explanations and knows nothing of windows or sound. `lineup-audio` plays a few notes apart from the song. `lineup-app` draws the window and joins the two.

A question from a number

Every question comes from a 64-bit **seed**:

```
pub fn new(settings: &IntervalSettings, seed: u64) -> Self {
    let mut random = Random(seed);
    let intervals = settings.intervals.places(Interval::ALL.len());
    let directions = settings.directions.places(Direction::ALL.len());
    let interval =
Interval::ALL[intervals[random.below(intervals.len())]];
    let direction =
Direction::ALL[directions[random.below(directions.len())]];
    let low = Note::new(INTERVAL_LOWEST + random.below(INTERVAL_SPAN) as
u8);
    IntervalQuestion {
        low,
        interval,
        direction,
    }
}
```

`Random` is the small SplitMix64 generator the twelve-tone rows shuffle with (see [What makes a row interesting](#)): the same seed always gives the same numbers, so the same question. That makes the tests simple, because they can ask thousands of questions and check each one, and a failing seed can be replayed exactly.

The window takes its first seed from the clock, and each next one from `next_seed`:

```
pub fn next_seed(seed: u64) -> u64 {
    Random(seed).next()
}
```

Why not just add the generator's own constant, `0x9E37_79B9_7F4A_7C15`? Because that is exactly what the generator does to itself for each number: the next question would start one draw further along the last one's numbers, so its first draw (a chord's kind) would be the last question's second (its root). After a C, an E or an A $\flat$ the next chord would always have been major. A seed taken from the generator's *output* starts fresh numbers; the test `one_question_does_not_give_the_next_away` asks 2,000 chords in a row and checks that the chord after a C can be any of the four kinds. If the

question is the same as the last one, the window tries again, up to eight times, so the same question rarely comes twice in a row.

A set of answers in sixteen bits

Which intervals or chords are asked is a set of places in a list. The list is short (twelve intervals at most), so the set is a `u16`, one bit per place, as a pitch-class set is twelve bits (see [Set theory in twelve bits](#)):

```
pub struct Choices(u16);
```

```
pub fn has(self, place: usize) -> bool {
    self.0 & (1 << place) != 0
}

/// Adds the answer at `place`, or takes it away. The last one left
/// stays: there is always something to ask.
pub fn toggle(&mut self, place: usize) {
    let toggled = self.0 ^ (1 << place);
    if toggled != 0 {
        self.0 = toggled;
    }
}
```

`^` (exclusive or) flips one bit. Refusing to flip the last one off means a question can always be asked, without a special case anywhere else.

Cues: what to play, in seconds

A question does not play itself: it returns **cues**, plain values that say who plays what, when and how loud.

```
/// One note of a question: who plays it, when, for how long and how loud.
#[derive(Debug, Clone, Copy, PartialEq)]
pub struct Cue {
    pub part: Part,
    /// When it starts, in seconds from the start of the question.
    pub start: f64,
    /// How long it is held, in seconds.
    pub length: f64,
    pub note: Note,
    pub velocity: u8,
}
```

Keeping sound out of `lineup-core` means the tests can check *what* would be played without playing it: that every bar of a progression plays its chord over its root, that the

piano alone adds the root, that a chord question's lowest note is the bass its symbol names.

The answers to compare with use the same idea. `cues_for(interval)` plays another interval from the same bottom note (or, going down, from the same top note), and `cues_as(quality)` another kind of chord on the same root with the same chord note in the bass. So when you click a wrong answer after answering, you hear exactly what you said next to what was played.

A phrase on the audio thread

The song is made of patterns and clips on a timeline. A question is not: it is a few notes that start *now*, on instruments of their own, while the song's channels stay as they are. So `lineup-audio` has a `Phrase`: its instruments and its notes, with their times in frames.

The audio thread must not allocate or free memory (see [The golden rule of the audio thread](#)). So the UI thread builds the whole phrase, instruments included, and sends it in one command:

```
PlayPhrase(Box<Phrase>),
```

The `Box` matters. A `Command` is as big as its biggest variant, and the test `commands_stay_small` keeps them at 80 bytes or less so the queue stays small. A `Phrase` has two `Vec`s and a few counters, around 80 bytes of its own; boxed, the command carries only a pointer.

On the audio thread the phrase only starts each note when its time comes and adds its instruments to the mix:

```

pub fn next_sample(&mut self) -> f32 {
    while let Some(note) = self.notes.get(self.next)
        && note.start <= self.clock
    {
        if let Some(instrument) =
self.instruments.get_mut(note.instrument) {
            instrument.note_on(note.pitch, note.velocity, note.gate,
self.sample_rate);
        }
        self.next += 1;
    }
    self.clock += 1;
    let mix: f32 = self
        .instruments
        .iter_mut()
        .map(Instrument::next_sample)
        .sum();
    mix * VOLUME
}

```

The notes are sorted by start, so `next` only ever moves forward: one comparison per sample when nothing starts. `while let ... && ...` is a let chain in a loop: it goes on while there *is* a next note *and* it is due. Each note is given its length (`gate`) when it starts, and the instrument lets go of it by itself, so the phrase does not need a list of notes to stop.

When a new phrase arrives, the old one goes back to the UI thread through the trash queue to be freed. A phrase that has played its last note waits two seconds, for the sound to ring out, and then goes the same way. `StopPhrase` (sent when you close the window or change tab) lets go of every note and drops the ones not started yet.

The instruments come from `build_instrument`, the same function the channels use: the piano is a channel with General MIDI program 0 (Acoustic Grand Piano), the bass program 33 (Electric Bass, finger), the drums the built-in kit. So with **SoundFont instruments** on (**Edit > Settings... > Audio**), the questions play the SoundFont's piano, bass and drums, like the band (see [SoundFonts and General MIDI](#)).

The window: actions out

The window does not hold the audio engine. It returns what it wants to hear as a list of actions, and the app plays them:

```
pub enum EarAction {
    /// Play these notes now, in place of anything it played before.
    Play(Vec<Cue>),
    /// Stop what it is playing.
    Stop,
}
```

Inside, the question and the answers given so far are one enum, `Asked`. Drawing the answers needs to change them *and* call methods of the window (to add to the score, to play a sound), which Rust does not allow while the answers are borrowed out of `self`. The window takes the question out of `self` for the time it is drawn, and puts it back:

```
match self.asked.take() {
    Some(Asked::Interval {
        question,
        mut answer,
    }) => {
        self.interval_answers(ui, &question, &mut answer, actions);
        self.asked = Some(Asked::Interval { question, answer });
    }
    // ... chords and progressions the same way
}
```

`Option::take` leaves `None` behind, so for those few lines `self` and the question are two separate values, and both can be borrowed freely.

The window's choices are remembered the way the other settings are: a `# [serde(default)]` struct kept in egui's memory, which eframe saves with the window's size.

Testing it

The metronome and the count-in are tested in the audio engine, frame by frame:

`the_count_in_comes_before_the_music`,
`the_count_in_runs_on_into_a_bar_started_in_the_middle` and
`stopping_ends_the_count_in`. The laps are tested twice:

`laps_count_the_loops_played_to_the_end` in the sequencer plays a four-step loop with jumps, restarts and a shortened loop, and
`laps_count_the_times_the_loop_came_round` and
`lap_tempos_start_on_the_loops_first_step` in the engine check the count and that a new speed starts exactly with the loop.

In the app, `the_trainer_moves_on_as_the_loop_comes_round`, `the_next_speed_is_asked_for_ahead_of_the_lap_it_starts_on` and `the_trainer_can_play_each_speed_more_than_once` drive the trainer with made-up lap counts, including a window that missed a few frames and an audio thread that restarted; `taps_set_the_tempo_of_their_beats` and `only_the_last_taps_count` check tap tempo, in 4/4 and in 6/8; and `the_trainer_moves_towards_its_target_and_stays_there` the speeds 70, 75, ... 100, a step that does not divide the way, and a target below the start.

The ear training core has 22 tests. The ones that guard the theory:

- **Every interval from every note** (`every_interval_is_spelled_as_its_name_says_from_every_note`): for all 12 bottom notes and all 12 intervals, the two written notes are the right keys, neither needs a double sharp or flat, and the name worked out from the two written notes is the interval's name.
- **Spellings by hand** (`intervals_are_written_the_simple_way`): C# E, D $\flat$ F, F B, E A#, C F#, E C, E $\flat$ D, F# F#, B $\flat$ D $\flat$, B $\flat$ E, A $\flat$ E $\flat$.
- **Chords against the analyzer** (`the_app_names_a_chord_in_root_position_as_the_question_does`): 500 chord questions in root position are fed to the analyzer the app uses to name what you play on a MIDI keyboard, which must name the same kind. Not inverted: the analyzer rightly hears Am7/C as C6.
- **Every chord question plays its chord with the right bass**, over 2,000 seeds with inversions on.
- **Spelled explanations** (`chords_are_spelled_from_their_root`): Bdim7, Ebmaj7/G, Fdim/C $\flat$, Cdim7's B $\flat\flat$, Caug/E (also Eaug), Ddim7/C $\flat$ (also Bdim7).
- **Numerals** (`numerals_are_those_of_the_harmony_keys`) match the harmony keys in major and minor, triads and sevenths.
- **Progressions** start on I, never repeat a chord, return to I only at the end, end on I or V, and use every chord chosen; every bar plays its own chord over its root.
- **Every key, root and bottom note comes up**: 24 keys, 12 roots, 16 bottom notes.

The phrase has four tests of its own in `phrase.rs` (notes start on time whatever order they come in, a phrase is over once it has rung out, stopping drops the notes still to come, notes for a missing instrument are left out), and three in the engine: a phrase plays with no song loaded and is thrown away after it rings out, a new phrase takes the place of the last, and a stopped phrase plays nothing more.

The lead sheet: from notes to notation

Files: `crates/lineup-core/src/lead_sheet.rs` and the modules in `crates/lineup-core/src/lead_sheet/` (`melody.rs`, `spelling.rs`, `rhythm.rs`, `score.rs`, `layout.rs`, `glyphs.rs`, `transpose.rs`), and in the app `crates/lineup-app/src/lead_sheet.rs` with `fonts.rs` and `pdf.rs`

A **lead sheet** is what a band reads from. It has one staff with the tune (the “lead”), the harmony written above it as chord symbols, and the song’s key, time signature, sections and repeats. It leaves out everything else: the bass line, the drums and how the piano voices its chords are for the players to make up, in the style of the evening. Jazz, pop and folk musicians carry whole books of them.

Ferrous Lineup holds everything a lead sheet needs: the tune is the notes of one channel, and the chords are the chord sheet’s (see [The chord sheet](#)). The **Lead sheet** window writes them out on pages you can print, as it sounds or for players of B $\flat$, E $\flat$ and F instruments. It only reads the song: it changes nothing in it or in the file.

Trying it

View > Lead sheet opens the window. The sheet is laid out on pages, as it will print, and the bar that plays lights up while the song plays. When the playing bar is out of sight, the window scrolls to it.

Lead sheet

Melody Auto (Lead) Clef Auto (treble) Paper A4 100% Export PDF... Print...

I Got Rhythm

$\text{♩} = 176$

5

9

13

17

Along the top:

- **Melody** picks the channel whose notes are the tune. **Auto** takes the channel called Lead, which every song demo has, or else the channel whose notes are highest on average, which is where a tune usually is; the menu shows which one, as "Auto (Lead)". Any channel with notes can be picked instead, the bass for a bass player's part. **None: chords only** writes no tune at all: each bar gets a slash for every beat, under the chord symbols, as a rhythm section reads its part.
- **Clef: Auto** writes a low tune in the bass clef and the others in the treble clef; **Treble** and **Bass** choose one.
- **For** writes the sheet for players of B $\flat$, E $\flat$ or F instruments (trumpet, saxophones, horn), in the key they read in (see [Parts for B \$\flat\$, E \$\flat\$ and F instruments](#) below). **Concert pitch** writes it as it sounds.
- **Paper:** A4 (used almost everywhere) or Letter (in the USA and Canada).
- – and + zoom the pages from 50% to 200%.
- **Export PDF...** saves the sheet as a PDF file.
- **Print...** writes the PDF to the computer's temporary folder and opens it in your PDF viewer, to print from there. Printing that way works the same on Linux, macOS and Windows, and the viewer shows the printer's own settings.

Lead sheet

Melody None: chords only ▼ Clef Auto (treble) ▼ Paper A4 ▼ - 100% + Export PDF... Print...

I Got Rhythm

♩ = 176

B $\flat$ Gm7 Cm7 F7 B $\flat$ Gm7 Cm7 F7

5 B $\flat$ B $\flat$ 7 E $\flat$ E $\flat$ m B $\flat$ F7 B $\flat$

9 B $\flat$ Gm7 Cm7 F7 B $\flat$ Gm7 Cm7 F7

13 B $\flat$ B $\flat$ 7 E $\flat$ E $\flat$ m B $\flat$ F7 B $\flat$

17 D7 G7

The paper, the zoom and the instruments under **For** are remembered for next time.

Reading a lead sheet

Here is the first page of La Cumparsita as the PDF prints it:

La Cumparsita

♩ = 112

Chords indicated above the staff:

- Staff 1: D7, Gm, D7
- Staff 2: Gm, Cm, Gm
- Staff 3: D7, Gm, D7, Gm
- Staff 4: D7, Gm, Cm
- Staff 5: Gm, D7, Gm, Gm, Em7^b5
- Staff 6: Gm, D7, Am7^b5, D7, Gm
- Staff 7: Em7^b5, Gm, Cm, Gm
- Staff 8: D7, Gm, Gm, Em7^b5, Gm, D7

1 / 2

From left to right on the first line:

- The **clef**. The treble clef curls around the G line (the second line from the bottom); the bass clef's two dots sit around the F line.
- The **key signature**: the sharps or flats of the key, so they need not be written on every note. Two flats (B $\flat$ and E $\flat$) are B $\flat$ major or G minor. La Cumparsita is in G minor, and its F $\sharp$ (the leading tone of the harmonic minor) gets a sharp each time, because the signature does not have it.
- The **time signature**, only on the first line and where it changes.
- The **tempo**: "♩ = 112", 112 quarter notes a minute. In 6/8 it is a dotted quarter, the beat you tap.

Above the staff are the **chord symbols**, each over the place where its chord starts. A chord is written once, when it changes; it goes on sounding until the next one. The small number at the start of each line is the **bar number** of its first bar, so a band can say "from bar 21".

Further on, a lead sheet can hold:

- **rehearsal marks**: the name of a section ("A", "Verse", "Chorus") in a box, where the section starts. Each section starts a new line, so the form of the song shows at a glance;
- **repeat signs**: a thick and a thin bar line with two dots, around a section played twice; "×3" over the end when it is played three times or more;
- a **double bar line** (two thin lines) before a new section and before a change of key, where the new key signature is written, with naturals for the sharps or flats the old key had and the new one does not;
- the **final bar line** (thin and thick) at the end.

From notes to notation

Notation is not a picture of the piano roll. The piano roll says "this key, from this step, for this many steps"; notation has to choose a letter for every note (A $\sharp$ or B $\flat$?), write every length as note values that can be counted, and show where the beats are. Musicians read fluently because these choices follow rules, and the rules are what this chapter is about. Each step is a module of **lineup-core**, and none of them knows about windows or PDF files.

The tune, one note at a time (melody.rs)

The notes of the melody channel are read along the playlist, clip by clip, as they play. A lead sheet has one note at a time, so where several notes start together the highest is

kept (the “skyline” rule: a tune harmonized in thirds is read as its top voice), and a note that is still sounding when the next one starts is cut short there.

Notes played a little early or late (recorded from a keyboard, or imported from a MIDI file) are moved to the nearest sixteenth. A lead sheet shows what is meant, not every nuance of a performance.

Which letter (spelling.rs)

The same key of the piano can be written two ways: the black key between G and A is G $\sharp$ or A $\flat$. Which one is right depends on what the note *does*, and the rules are tried in this order:

```

pub fn spell(key: Scale, pitch: Note, chord: Option<ChordSymbol>, next:
Option<Note>) -> NoteName {
    let pitch_class = pitch.pitch_class();
    if key.has_letter_for(pitch_class) {
        return key.spell(pitch_class);
    }
    if let Some(symbol) = chord {
        let names = symbol.chord().spell(symbol.root);
        let bass = symbol.bass.into_iter();
        if let Some(name) = names
            .into_iter()
            .chain(bass)
            .find(|name| name.pitch_class() == pitch_class &&
name.accidental.abs() <= 1)
        {
            return name;
        }
    }
    if key.kind.is_tonal()
        && let Some(next) = next
    {
        let step = i32::from(next.midi()) - i32::from(pitch.midi());
        let neighbour = match step {
            1 => Some(key.spell(next.pitch_class()).letter.up(-1)),
            -1 => Some(key.spell(next.pitch_class()).letter.up(1)),
            _ => None,
        };
        if let Some(letter) = neighbour {
            let name = NoteName::with_letter(pitch_class, letter);
            if name.accidental.abs() <= 1 && !is_odd(name) {
                return name;
            }
        }
    }
    let name = key.spell(pitch_class);
    if is_odd(name) {
        NoteName::chromatic(pitch_class)
    } else {
        name
    }
}

```

1. **A note of the key** is written as the key writes it: in A major, the black key above C is C#, never Db.
2. **A note of the chord** that sounds with it is written as the chord spells it. Over E7 in C major the black key above G is G#, the major third of E (E G# B D). Over Fm in C major the same key is Ab, the minor third of F (F Ab C). The letters follow the chord's thirds.
3. **A chromatic neighbour**, a note that moves a semitone to the next one, takes its letter from the way the line goes: up, it is the raised letter below (F# going to G);

down, the lowered letter above (B $\flat$ going to A). The note “leans” towards where it goes.

4. **Anything else** gets the name the key gives notes outside it.

`is_odd` keeps outside notes plain: outside the key, a white key of the piano is written as one. In a key with many flats, a note B between C and B $\flat$ is written C – B – B $\flat$, not C – C $\flat$ – B $\flat$, which reads worse even though C $\flat$ is the “same” note.

Once each note has its letter, the bar decides which **accidentals** to print. An accidental lasts until the bar line: in G minor, the first F $\sharp$ of a bar gets a sharp, the next F $\sharp$ in that bar does not, and an F after it gets a natural. The next bar starts again from the key signature.

Rows and pitch-class sets have no key signature, so a song in a row or a set is written without one, every black key with its own sharp or flat.

How long (rhythm.rs)

A note four sixteenths long is a quarter note, wherever it is. But the reader counts beats, and notation must show them. So every place in a bar has a **strength**:

```
/// How strong a place in a bar is, counted in sixteenths from its start:
///
/// - 4 for the bar line;
/// - 3 for the middle of a bar of four even beats (beat 3 of 4/4, the
///   third dotted quarter of 12/8);
/// - 2 for any other beat;
/// - 1 for the halves of a beat (the "and" of 4/4), or the thirds of a
///   dotted beat (each eighth of 6/8);
/// - 0 for anything smaller.
pub fn strength(time: TimeSignature, offset: usize) -> u8 {
```

The rules, as engravers use them (they are written down in Elaine Gould’s *Behind Bars*, the reference book of music engraving), say which notes may cross which places:

- A note may hold over a beat only if it starts on a beat. A quarter note that starts on the “and” of beat 2 in 4/4 hides beat 3, so it is written as two eighths **tied** together: the second note is not played again, the tie (a thin curve) holds the first one on.
- Nothing hides the middle of a bar of four beats, unless it starts at the bar line. A note from beat 2 to the end of the bar is a quarter tied to a half.
- One pattern is let through because every reader knows it: the syncopation “♪ ♩”, a quarter from the middle of one beat to the middle of the next, inside one half of the bar.

- In 6/8 and in grouped meters like 2+2+3/8 notes hold only from beat to beat, so the groups stay visible.
- Rests are stricter still: a rest covers several beats only from the bar line or the middle of the bar, as a half rest on beat 1 or 3 of 4/4.

When a note does not fit in one value, it is cut at the strongest place inside it first, and the pieces are tied. Here are four bars of I Got Rhythm: the tune's syncopations come out as quarters and eighths tied over beat 3, the middle of the bar, so the reader always sees where it is.



Beams join eighths and sixteenths beat by beat (by the whole bar when the beats are themselves eighths, as in 3/8), so the beats show here too. A beam slants with the notes, but never by more than one staff space, and a sixteenth beside an eighth gets a short second beam of its own, a “stub”.

What a bar holds (`score.rs`)

`score.rs` puts these together bar by bar. A bar holds its notes and rests (each with its value, letter, accidental and tie), the chords that start in it, and whatever is written at its edges: a new key or time signature, the start of a section, repeat signs. Before the first note of the tune and after the last one, a bar where the band plays a chord gets slashes instead of rests, so an introduction or an ending the band plays alone reads as “play these chords”.

Where everything goes (`layout.rs`)

Then everything is placed on the page, in points (1/72 of an inch). The staff is 21.2 points high (7.5 mm), four **staff spaces** of 5.3 points, the usual size for a lead sheet. Every size in the layout is counted in staff spaces.

Notes get room by how long they last, but not in proportion: a half note does not get twice the room of a quarter note, or a bar of sixteenths would be crushed next to a bar of whole notes. Engravers space by something close to the square root of the length:

```

/// How much room a note or rest gets for lasting `steps` sixteenths,
/// before the line is stretched: about 2.6 spaces for a sixteenth, 4.2
/// for a quarter, 7.4 for a whole note.
fn spring(steps: usize) -> f32 {
    sp(1.0 + 1.6 * (steps as f32).sqrt())
}

```

A chord symbol is wider than a note, so a quick chord change (“Gm G7” over two quarter notes) pushes its notes apart until the symbols do not touch.

A line holds up to four bars, fewer if they do not fit, and a new section starts a new line. Each full line is then **stretched** until it reaches the right margin exactly. How much to stretch is found by trying: draw the line with a stretch of 0.25 and of 6, then halve the gap 24 times, keeping the half where the width lands. This is a **binary search**, the same idea as finding a word in a dictionary by opening it in the middle:

```

fn fit(&self, bars: Range<usize>, width: f32) -> f32 {
    let wide = |stretch: f32| self.draw_system(bars.clone(),
stretch).width;
    let (mut low, mut high) = (0.25_f32, 6.0_f32);
    // ...
    for _ in 0..24 {
        let middle = (low + high) / 2.0;
        if wide(middle) > width {
            high = middle;
        } else {
            low = middle;
        }
    }
    low
}

```

`wide` is a **closure**, a small function written inline that borrows `self` and `bars` from around it. The line is drawn up to 26 times to place it once; it is quick, because drawing a line only computes positions.

The last line of a section or of the song is often short. Stretched to the margin, its few notes would sit far apart, so it is spaced like the full lines and left short, as engravers do. Lines then go on pages, spread out so each page is filled from top to bottom, except the last.

One list of marks

The layout’s result is a list of **marks** per page: plain values that say what to draw where.

```
pub enum Mark {
    /// A symbol of the music font with its origin (the SMuFL baseline)
    /// at `at`; `size` is the font size, the height of a staff.
    Music {
        glyph: glyphs::Glyph,
        at: Point,
        size: f32,
    },
    /// Text with the left end of its baseline at `at`.
    Text {
        text: String,
        font: TextFont,
        at: Point,
        size: f32,
    },
    /// A straight line `width` thick, with square ends.
    Line { from: Point, to: Point, width: f32 },
    // ... Beam and Tie, each with two points and a thickness
}
```

The window paints these marks on screen and `pdf.rs` writes the same marks to the PDF, so what is printed is exactly what was shown. This is the same split the app uses everywhere: `lineup-core` decides, the app only draws. It also makes the layout testable without a window: a test can lay out a song and look for a notehead per note, a box per bar, and nothing outside the page.

The fonts

Music symbols come from a **music font**. Fonts that follow **SMuFL**, the Standard Music Font Layout, give every symbol a fixed code point (the treble clef is U+E050 in all of them) and are drawn so that one em, the font's size, is the height of the staff. The app uses **Bravura**, the font SMuFL was designed with, by Steinberg. The words and chord symbols are in **Inter**, a font made for screens that prints cleanly too. Both are free, under the SIL Open Font License, and are built into the program with `include_bytes!`, so the sheet looks the same on every computer:

```
const BRAVURA: &[u8] = include_bytes!("../assets/fonts/Bravura.otf");
```

`include_bytes!` is a macro that reads the file when the program is compiled and puts its bytes in the program, as a `&'static [u8]`: a slice that lives as long as the program does.

The layout needs to know how wide each symbol is and how far it reaches up and down. Those numbers come from Bravura's own metadata, copied into `glyphs.rs`,

and a test reads the font file to check every one of them.

Text is measured with the very font it is drawn with. `lineup-core` has no fonts, so it asks through a **trait**:

```
pub trait TextWidth {  
    /// The width of `text` in `font` at `size` points, in points.  
    fn text_width(&self, text: &str, font: TextFont, size: f32) -> f32;  
}
```

The app implements it by adding up the letters' widths read from the font file with the `ttf-parser` crate; the tests implement it with a simple rule (every letter half as wide as the font size). The layout takes a `&dyn TextWidth` and works with either.

Chord symbols follow the custom of lead sheets: the root letter large, the rest a little smaller ("Am7", "G7"), and real ♯ and ♭ signs from the music font, raised, instead of "#" and "b". The roots are **letters whatever the app's note naming**: with solfège chosen, the chord sheet says Rem7, but the lead sheet writes Dm7, as printed lead sheets do in every language, so a band can read it anywhere.

The PDF

A PDF page is a list of drawing operations, much like the marks: move to a point, draw a line to another, fill a shape. The `pdf-writer` crate writes them in the PDF's format; it runs on every platform and needs nothing from the system.

Letters and symbols are not embedded as fonts. Each one is copied in as its **outline**, the curves that draw it, read from the font file with `ttf-parser`, and filled. A symbol used many times, like a black notehead, is written once as a "form XObject" and then placed wherever it appears, scaled to its size:

```
content  
    .save_state()  
    .transform([scale, 0.0, 0.0, scale, x, height - at.y])  
    .x_object(Name(name.as_bytes()))  
    .restore_state();
```

The `transform` is a matrix that scales the shape from the font's units to points and moves it to its place. Notice `height - at.y`: PDF counts y upwards from the bottom of the page, the layout counts it downwards from the top, so every y is turned round. The PDF looks the same in every viewer, with nothing to install.

Parts for B $\flat$, E $\flat$ and F instruments

Put a lead sheet of a song in C in front of a trumpet player and a pianist, ask them to play the first note together, and they will not agree. The pianist plays C; the trumpet, reading the same C, sounds the B $\flat$ a whole tone lower. The trumpet is a **transposing instrument**: the notes it reads are not the notes it sounds. To play with the band, the trumpet player needs a sheet written a whole tone higher, in D.

Why some instruments read in other keys

Wind instruments come in families of different sizes: the saxophones from soprano to baritone, the clarinets, the horns. Each size sounds lower or higher, but the fingers are the same: on every saxophone, the same fingers play the note written C. If the music were written as it sounds, a player switching from alto to tenor would have to learn new fingerings for every note. So the music is written for the fingers instead, and each instrument is named after the note that sounds when it plays a written C: a trumpet **in B $\flat$** , an alto saxophone **in E $\flat$** , a horn **in F**.

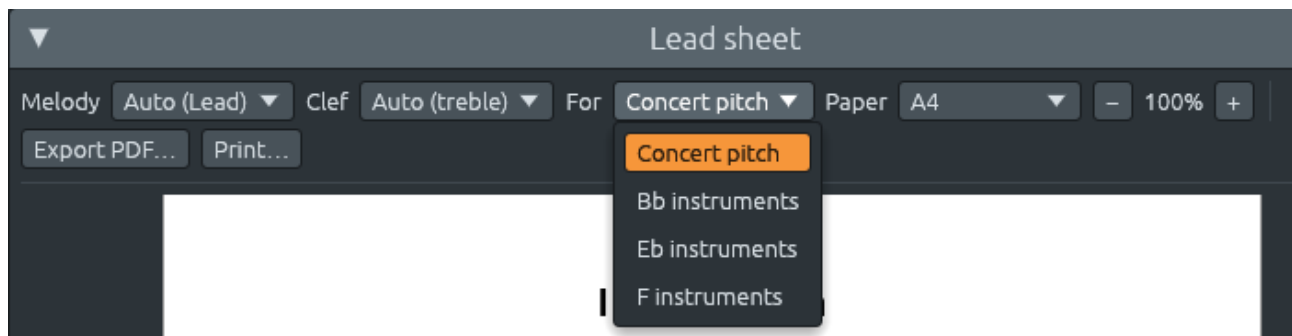
Music written as it sounds is at **concert pitch**, or “in C”. Piano, guitar, bass, voice, flute, violin and trombone all read it.

Instruments	A written C sounds	The part is written
Piano, guitar, bass, voice, flute, violin, trombone	C	as it sounds
Trumpet, cornet, flugelhorn, clarinet, soprano saxophone	the B $\flat$ below	a major second higher
Tenor saxophone	the B $\flat$ an octave and a second below	a major ninth higher
Alto saxophone	the E $\flat$ below	a major sixth higher
Baritone saxophone	the E $\flat$ an octave and a sixth below	an octave and a sixth higher
Horn, English horn	the F below	a perfect fifth higher

The part is always written higher than it sounds, and within a family by the same interval give or take an octave, so a song in C is written in D for every B $\flat$ instrument, in A for every E $\flat$ instrument and in G for every F instrument.

Using it

In the Lead sheet window, **For** offers:



- **Concert pitch** writes the song as it sounds.
- **Bb instruments**: trumpet, clarinet, soprano and tenor saxophone.
- **Eb instruments**: alto and baritone saxophone.
- **F instruments**: horn and English horn.

Hover over a choice to see which instruments it is for and how much higher it is written. The sheet changes at once: the key signature, the tune and the chord symbols move together, and the top left corner of the first page says who the part is for, “B $\flat$ instruments”, where printed parts name their instrument. **Export PDF...** offers the file name with the instruments in it, *I Got Rhythm (Bb instruments).pdf*, so the parts of one song do not overwrite each other. The choice is remembered: a saxophone player who opens another song finds it written for them again.

Here is the I Got Rhythm demo for B $\flat$ instruments. The song is in B $\flat$ major, two flats; the part is in C major, with no sharps or flats at all. This is one reason so many jazz tunes are in flat keys: concert B $\flat$ is C for the trumpet and the tenor saxophone, and concert E $\flat$ is C for the alto saxophone.

Lead sheet

Melody Auto (Lead) Clef Auto (treble) For Bb instruments Paper A4 100%

Export PDF... Print...

Bb instruments **I Got Rhythm**

$\text{♩} = 176$

The demo starts with an introduction on G minor chords (Gm, D7, Cm). On the B $\flat$ part they are Am, E7 and Dm; on the E $\flat$ part, in G major, Em, B7 and Am; on the F part, in F major, Dm, A7 and Gm. The song itself, its file and what the band plays are left as they are. Only the sheet is written differently.

The key: never more than six sharps or flats

Moving the key is adding semitones: two for B $\flat$ instruments (B $\flat$ up to C), nine for E $\flat$ instruments (E $\flat$ up to C) and seven for F instruments (F up to C).

```
pub fn semitones(self) -> u8 {
    self.note().pitch_class().semitones_up_to(PitchClass::C)
}
```

```
pub fn key(self, key: Scale) -> Scale {
    Scale::new(key.root.transpose(i32::from(self.semitones())), key.kind)
}
```

`note()` is the note a written C sounds, B $\flat$ for `Transposition::BFlat`, so the instruments' names and their distances come from one place. The kind of key stays: F minor is G minor for a trumpet, D Dorian is B Dorian for an alto saxophone.

The new key is a `Scale` like any other, so it is named the way the app names every key (see [Sharps or flats: how a key is written](#)): each of the twelve keys has one name with at most six sharps or flats, and that is the one used. This matters for parts, because counting up by letters would sometimes give keys nobody writes in:

Song	B $\flat$ part	E $\flat$ part	F part
C	D	A	G
F	G	D	C
B $\flat$	C	G	F
E $\flat$	F	C	B $\flat$
A $\flat$	B $\flat$	F	E $\flat$
D $\flat$	E $\flat$	B $\flat$	A $\flat$
G	A	E	D
D	E	B	A
A	B	F $\sharp$	E
E	F $\sharp$	D $\flat$	B
B	D $\flat$	A $\flat$	F $\sharp$
F $\sharp$	A $\flat$	E $\flat$	D $\flat$

A song in E for an alto saxophone, a major sixth up, would strictly be in C $\sharp$ major, seven sharps. D $\flat$ major has the same notes with five flats, and it is what a copyist writes. A song in F $\sharp$ for a trumpet would be in G $\sharp$ major, which needs an F double sharp in its key signature; A $\flat$ major, four flats, has the same notes. A test goes through all twelve major and minor keys for every instrument and checks that no part has more than six.

The chords: moved letters and all

A chord must keep its place in the key: the V7 of the song is the V7 of the part. A chart is moved to another key by counting letters as well as semitones (see [Moving a chart to another key](#)), and the part uses the same function, from the song's key to the part's:

```
pub fn harmony(self, harmony: HarmonyName, key: Scale) -> HarmonyName {
    harmony.moved(key, self.key(key))
}
```

Because it goes from tonic to tonic, it follows the key's spelling too. From C to D every letter moves up one, so Dm7 G7 Cmaj7 becomes Em7 A7 Dmaj7. From F $\sharp$ to A $\flat$ every letter moves up two, so the V7 of F $\sharp$, C $\sharp$ 7, becomes E $\flat$ 7, the V7 of A $\flat$, and never D $\sharp$ 7.

The bossa nova demo Garota de Ipanema is in D $\flat$; for an alto saxophone, in B $\flat$, its D $\flat$ maj7 Eb7 Ebm7 D7 are B $\flat$ maj7 C7 Cm7 B7.

The notes: every interval keeps its name

The tune moves by the same semitones, but its notes need letters too, and here a short cut would go wrong. The obvious way is to move each note up and then spell it again in the new key with the rules of [Which letter](#). Mozart's Turkish March, in A minor, shows the problem. Its tune leans on E from the semitone below, D $\sharp$ -E. In B minor, the trumpet's key, the same sounds would be spelled again as F-F $\sharp$: the rules keep a white key of the piano plain outside the key, so they write F rather than E $\sharp$. But D $\sharp$ -E is a minor second, a note leaning on the next, and F-F $\sharp$ is an augmented unison, one letter bent upward. A player reads a different line.

So the notes are spelled at concert pitch first, where the rules know the key and the chords the song was written in, and then each name is moved by the same letters and semitones as the key:

```
pub fn name(self, name: NoteName, key: Scale) -> NoteName {
    let written = self.key(key);
    if written == key {
        return name;
    }
    if !key.as_key().kind.is_tonal() {
        return
written.spell(name.pitch_class().transpose(i32::from(self.semitones())));
    }
    let (from, to) = (key.spell(key.root), written.spell(written.root));
    let moved = name.above(
        from.pitch_class().semitones_up_to(to.pitch_class()),
        from.letter.steps_up_to(to.letter),
    );
    if moved.accidental.abs() >= 2 && written.spell(moved.pitch_class())
!= moved {
        moved.simplified()
    } else {
        moved
    }
}
```

`NoteName::above` is the same function that moves chord roots, so the notes and the chords move by exactly the same letters. The D $\sharp$ -E of the Turkish March is E $\sharp$ -F $\sharp$ for a trumpet, B $\sharp$ -C $\sharp$ for an alto saxophone and A $\sharp$ -B for a horn: a minor second each time, as Mozart wrote it. A row or a set has no letters of its own (the second `if`), so its notes take their usual names in the new key.

The last lines deal with the keys that were renamed. A song in E has a lowered sixth, C. For an alto saxophone, in D $\flat$ rather than C $\sharp$, the letters move from E to D, and C becomes B double flat (B $\flat\flat$), the lowered sixth of D $\flat$ spelled strictly. Nobody wants to read that, and it is the same note as A, so the name is made simpler, as chord symbols are. A double sharp or flat that the new key itself has stays: the leading tone of C $\sharp$ harmonic minor is B $\sharp$, and on the trumpet part, in D $\sharp$ harmonic minor, it is C double sharp (C $\sharp\sharp$), the leading tone of that key.

The octave: where the tune sits on the staff

The interval says which notes to write, not in which octave. A tenor saxophone sounds a ninth below its written notes, an octave lower than a trumpet, yet in books of lead sheets for B $\flat$ instruments both read the same page: the tenor player simply sounds it an octave lower. What matters on the page is that the tune sits on the staff, not on a ladder of leger lines above it.

So the part is written by the instrument's interval, moved by whole octaves until the middle note of the tune is as near as it can be to the middle line of the staff (B4 in the treble clef):

```
pub fn shift(self, pitches: &[Note], clef: Clef) -> i32 {
    let interval = i32::from(self.semitones());
    if self == Transposition::Concert {
        return 0;
    }
    let mut sorted = pitches.to_vec();
    sorted.sort();
    let Some(median) = sorted.get(sorted.len() / 2) else {
        return interval;
    };
    let middle = i32::from(clef.middle_line().midi());
    let median = i32::from(median.midi());
    (-4..=4)
        .map(|octaves| interval + 12 * octaves)
        .min_by_key(|shift| ((median + shift - middle).abs(), (shift -
interval).abs()))
        .unwrap_or(interval)
}
```

The middle note is the **median**, the one in the middle once the notes are sorted, as for choosing the clef: a few very high or low notes do not pull it away. For a tune in the range of a voice the result is the instrument's own interval: a major second up for B $\flat$ instruments, a major sixth up for E $\flat$ instruments, a fifth up for F instruments, so trumpet, alto saxophone and horn sound the tune where it is. A tune around C6

comes out a minor seventh *lower* for B $\flat$ instruments, the same letters an octave down, its middle note on D5, where a trumpet can play it.

`min_by_key` keeps the candidate with the smallest key, and the key here is a pair. Pairs compare like words in a dictionary: by the first item, and by the second only when the first ones are equal. So the distance to the middle line decides, and when two octaves are equally near, the one nearer the instrument's own interval wins.

Parts for transposing instruments use the treble clef, which all of them read; **Clef** can still choose the bass clef, and then the tune is centred on that staff's middle line, D3.

The screenshot shows a software window titled "Lead sheet" with a close button in the top right. Below the title bar is a control bar with several dropdown menus and buttons. The "Melody" dropdown is set to "Auto (Lead)". The "Clef" dropdown is set to "Auto (treble)". The "For" dropdown is set to "Eb instruments". The "Paper" dropdown is set to "A4". There are also buttons for "Export PDF..." and "Print...". The main area of the window displays musical notation for a piece titled "Turkish March". The notation is in treble clef with a key signature of three sharps (F#, C#, G#) and a time signature of 4/4. The tempo is marked as "♩ = 126". The notation is divided into four systems, each with a measure number (1, 4, 7, 10) at the beginning. Chord symbols are placed above the notes: F#m, F#m, C#m, G# C#, F#m, C#m, G# C# A Bm, A, E, A, E, F#m, C#. The notation includes various note values, rests, and bar lines.

In the code

The lead sheet is split into steps, and only some of them know about instruments:

- `transpose.rs` holds the `Transposition` type with `key`, `harmony`, `name` and `shift`.
- `score.rs` asks it for the key of each bar, the chords, the names of the notes and the octave. Two closures keep that short:

```
let key_at = |step: usize| transposition.key(keys.key_at(step));
let harmony_of =
    |marker: ChordMarker| transposition.harmony(marker.harmony,
        keys.key_at(marker.start));
```

They borrow `keys` and `transposition` from the function around them, so each call only says which step or which marker.

- `layout.rs` writes “B♭ instruments” at the top left of the first page, with the flat from the music font as in chord symbols. When the title is too wide to leave room beside it, the words go on a line under it.
- `melody.rs` and `rhythm.rs` know nothing of instruments: the tune is read, spelled at concert pitch, rounded and cut into note values the same way for every instrument.

In the app, the transposition is stored with the paper and the zoom in the window’s settings:

```
#[derive(Debug, Clone, PartialEq, Serialize, Deserialize)]
#[serde(default)]
struct Settings {
    paper: Paper,
    zoom: f32,
    /// Who reads the sheet: a player keeps their instrument from song to
    /// song.
    transposition: Transposition,
}
```

`#[serde(default)]` lets a field arrive without breaking anything: settings saved without a `transposition` get it from `Settings::default()`, concert pitch. `Transposition` itself gets its default from an attribute on a variant:

```
#[derive(Debug, Clone, Copy, PartialEq, Eq, Default, Serialize,
    Deserialize)]
pub enum Transposition {
    /// Written as it sounds: piano, guitar, bass, voice, flute, violin,
    /// trombone and every other instrument "in C".
    #[default]
    Concert,
    // ... BFlat, EFlat and F
}
```

The project file does not change: who reads the sheet is a choice of the reader, not part of the song.

Testing it

`lineup-core` tests each step of the lead sheet on its own:

- **spelling** (`spelling.rs`): the key's letters; a chord's letters (G# over E7 and A $\flat$ over Fm in C major); chromatic steps by direction; white keys outside the key stay plain; minor keys raising their leading tone; accidentals that last to the bar line; key signatures in both clefs; the naturals of a key change;
- **rhythm** (`rhythm.rs`): the strength of every place; notes on the beat holding over beats; the middle of a bar of four; a note off the beat not hiding the next beat; the "♪ ♪ ♫" syncopation left untied; compound beats kept whole; uneven bars split at their beats; rests showing every beat; beams by the beat; every piece a written value;
- **melody and score** (`melody.rs` , `score.rs`): the top note of a chord as the tune, notes off the grid on the nearest step, a repeated clip played again, notes tied over the bar line, the bass clef for low tunes, slashes or rests in bars without a tune, rehearsal marks, repeats and key changes;
- **layout** (`layout.rs`): a notehead for every note and a box for every bar, everything on the page, a section starting a new line, chords alone over slashes, chord symbols that never run into each other and use real accidentals in either note naming, beams and ties, repeats, and a part that says who reads it, beside or under the title;
- **parts** (`transpose.rs` and `score.rs`): every instrument's interval; a song in C written in D, A and G, minor keys and modes keeping their kind; no part key with more than six sharps or flats, for all twelve major and minor keys and every instrument; chords keeping their place (ii-V-I in C is Em7 A7 Dmaj7, C#7 in F# is E $\flat$ 7 in A $\flat$, a polychord and N.C. too); notes keeping their intervals (D#-E in A minor becomes E#-F#, B#-C# and A#-B; the lowered sixth of E is A in D $\flat$; C## stays in D# harmonic minor); the octave for a tune around G4, a high tune and a bass line, in both clefs; a trumpet part in D major; a key change in the parts' keys; the treble clef for parts.

The app has four: every symbol of the music font is in Bravura at the size the layout thinks it is, text is measured with the font, a song becomes a PDF with a page object per page and its shapes written once each, and a part is saved under a name of its own, `Op. 24 (Bb instruments).pdf`.

Tests cannot tell whether a sheet reads well, so the sheets are also read as a musician reads them: Ode to Joy, Für Elise, Minuet in G, Careless Love, Korobeiniki, Funiculì, funiculà, The Entertainer, the Moonlight Sonata, El Choclo, Greensleeves, La Cumparsita, When the Saints Go Marching In and I Got Rhythm at concert pitch, and I

Got Rhythm, the Turkish March, with its many changes of key, and Garota de Ipanema as parts.

Exercises: the band

These exercises practise this part on the real code: the library, the chord sheet, the band and its players, the orchestra, the song's form, the stops, the soloist, your own styles, the practice tools and the lead sheet. Each one says what is **already there**, so you extend it rather than build it again. Some are things to try in the app and listen to; some are pieces of Rust with a clear end; the ones called "Think about it" are questions, with no code to write.

Try the Rust ones on a branch of your own. Run `cargo test` after each change, and `cargo fmt --all --check` and `cargo clippy --workspace --all-targets -- -D warnings` before you call one finished: CI runs all three. An exercise that adds something to the project file needs the next free format number (see [The version number](#)).

The library

1. A style of your own (lineup-core, easy)

`STYLES` in `library/style.rs` is a table of `style(...)` rows (see [Styles: a table in a static](#)). Add one: a progression from a song you like, written in numerals. For example the whole of Pachelbel's Canon, `I | V | vi | iii | IV | I | IV | V` (the **Piano ballad** style has only its first six chords), with `Comping::ArpUpDown` and `Bass::Held`. Run `cargo test -p lineup-core` before you open the app: if a numeral is misspelled, a bar cannot hold its chords or the name is taken, `every_style_pares_and_fits_in_a_pattern` or `names_are_unique_and_rhythms_exist` names the style.

Then try it in the app over a song in another key. Which chord names change, and which numerals stay the same?

Practises: `const fn` tables, struct literals, reading test failures.

2. A rhythm with a cowbell and a fill (lineup-core, easy)

Add a drum pattern to `RHYTHMS` in `library/rhythm.rs`, in its genre's block, with a line for `Kick`, one for `Snare` and one for `Cowbell`, and a fill that exists, such as `"Tom`

fill". In 4/4 every line is sixteen characters, from `x`, `x`, `o`, `h`, `l` and `.`; spaces are only for reading. Put the cowbell on an offbeat or a clave, not on every step, so it is heard as a colour of its own.

Done when `every_line_is_whole_bars_of_known_characters` and `names_are_unique_and_fills_exist` pass, the Library lists the rhythm with its sentence, and, with the band playing it, the fill comes in the last bar of every eight.

Practises: data as text, tests that read the whole table.

3. A third chacarera (lineup-core, easy)

The library has a `Chacarera` rhythm in 6/8 and a `Chacarera trunca` in 3/4. Read both first: a bar of each is twelve steps. Which steps start the beats of a 6/8 bar, and which those of a 3/4 bar? Now write a third bombo pattern for the same dance, with the head (`LowTom`) on the 6/8 beats and the rim (`Rim`) on the 3/4 ones, the two against three of the genre, and a style that uses it. Check in the rack that the beat colours come in two groups of six.

Practises: reading existing data before adding to it, meters with the same number of steps.

4. Near the song's tempo (the app, easy)

In the Library, on a song at 60 BPM, tick **Near 60 BPM**. Which tempos does it keep? Then on a song at 180. `NEAR_SPEED` is 1.15: work the ranges out before you look, and say why a fixed ± 15 BPM would have been worse (see [The Library tab](#)).

Practises: ratios.

5. The same genre (lineup-app, medium)

Add a fourth box, **Same genre as the song's style**, that keeps the presets of the genre the band plays. What does `Narrowing` need to know, where do you get it from, and what should the box do in a song without a band?

Practises: adding a filter without touching the others.

6. Think about it: closures that run once

`write_pattern_style` takes `write: impl FnOnce(&mut Project, usize)`. The two closures passed to it today only read what they capture. Would `impl Fn` work for them? What could a later closure do that `Fn` would refuse, and why is asking for `FnOnce` the kindest choice for whoever writes the next one?

The chord sheet

7. How many bars fit (pencil, easy)

`Columns::count` (see [Why 1, 2, 4, 8, 12 or 16](#)) gets the width of the sheet and the width the widest bar needs. Work out by hand how many bars a line holds in a sheet 900 pixels wide when the widest bar needs 100, 120 and 250 pixels. Then add those three cases to the test next to `count`, and run it.

Practises: reading a short function closely.

8. Insert several bars (lineup-app, easy)

Insert a bar before and **Insert a bar after** call `project.insert_bars(bar, after, 1)`. Add **Insert bars...**, which asks how many. Which line of `open_bars` already makes room for more than one, and what stops a song from growing past its longest?

Practises: using what the core already offers.

9. A bar inside a repeat (lineup-core, medium)

Write a test in `bars.rs`: a clip repeated three times, and a bar inserted in the middle of its second time round. Write down first which clips, at which steps, you expect afterwards; then run it. The tests

`a_clip_of_a_pattern_played_elsewhere_too_is_cut_in_two` and `a_repeated_clip_loses_the_repeats_in_the_deleted_bars` show how to set one up.

Practises: writing the expectation before the code tells you.

10. Think about it: before or after

`close_bars` reads the time signature, the tempo and the key at the end of the deleted bars *before* it moves anything (see [What a deleted bar takes with it](#)). What would it read if it asked afterwards, and what would the song sound like?

11. Two bars again (lineup-core, medium)

`%` plays the bar before again (`repeat_bar`). Charts have a sign for the two bars before as well, a `%` with two slashes, drawn across the bar line of the two bars it fills. Let `%%` typed in a bar repeat the two bars before it, in that bar and the next, which must be empty. `read_typed_bar_after` reads one bar and is given one bar before: what must it be given now, and how does the sheet window learn that typing in one bar wrote two? What should `text::write` write for two bars that repeat the two before?

Practises: a notation that spans more than the place it is typed in.

12. A title in the song (lineup-core and lineup-app, medium)

`text::read` reads a `Title:` line into `TextSheet::title` , but the song does not keep it: **Export chord sheet...** writes the song's file name as the title. Keep the title in the project and write it back. It is a new field of the file: what does an older file read as?

Practises: a new field from the text format to the project file.

13. Paste in the song's key (lineup-app, medium)

A chart pasted from a message keeps its chord names, even when its `Key:` line says another key than the song's. Add **Paste in the song's key**, which moves the pasted chords from the chart's key to the song's. `HarmonyName::moved(from, to)` already does it for one harmony, keeping letters right. What does the paste need to know that it does not keep today?

Practises: reusing a theory function in an editing action.

14. Bars per line in text (lineup-core, easy)

`text::write` puts at most four bars on a line, `BARS_PER_LINE` . Make it a parameter. What in the column layout assumes four, and how does `widths` have to change?

Practises: turning a constant into a parameter, arrays and vectors.

15. Think about it: other chart formats

- ChordPro, a common format for songbooks, writes chords inside the lyrics:
`[C]Twinkle twinkle [F]little [C]star`. It has no bar lines. What would a reader need to guess to make a `TextSheet` from it?
- Some players want the numeral large and the chord name small under it.
`HarmonyName::caption` gives the numeral of a chord. What should the large line say for a set, which has no numeral?

How the band plays

16. Bass lines by eye (the app, easy)

Type `C | Am | F | G7 | C` and pick **Jazz waltz**. Look at the bass in the piano roll: which note does it play on beat 3 of each bar, and how far, by name, is it from the next bar's root? Then pick **Swing**, in 4/4: what does the walk put on beats 2 and 3 now? Check your answers against `walk` in [The bass](#).

Practises: reading a bass line.

17. Where the chord changes (the app, easy)

Type a bar `Dm7 G7` and a bar `C / / G7` in **Pop beat**. On which beat does G7 start in each? Look at the piano part: does its rhythm change in those bars, or only its notes?

18. Alberti bass (the app, easy)

Pick **Alberti bass** over `C | Am | F | G7`. The pattern is "low, high, middle, high". Which note is low in the Am bar, and why is it not A? (Hint: look at all four bars, and at [The harmonies, as notes](#).)

19. A part you changed (the app, easy)

In a new song, change one note of the bass, then type a new chord in bar 2. Which parts follow the new chord, and which doesn't? What does the notice say? Click **Play it again**: what happens to your note?

20. A band over a demo (the app, easy)

Open **Canon in D** and click **Pop ballad** in the Library. Look at the channel rack: which channels are muted, and which play on? Then pick **No band** in the band's menu. Is the demo back exactly as it was?

21. Think about it: what a fingerprint leaves out

- `part_print` leaves out the track a clip is on and its pattern's number. What would the user see if it did not?
- `band_styles()` returns `&'static [BandStyle]`. Why can `Band` hold a `&'static BandStyle` without a lifetime parameter, when it could not hold a `&BandStyle` borrowed from a local `Vec`?

A bigger band, horns, and following your chords

22. Four guitarists (the app, easy)

Make a chord sheet with `C Am F G` and try **Pop beat**, **Folk**, **Rock beat** and **Swing**. Click **Piano**, **Bass** and **Drums** in the **You play** row, so that only the guitar is left. How does each guitarist play the same four chords?

23. Chord shapes (pencil and the piano roll, easy)

In **Pop beat**, read the guitar's notes for G and for F in the piano roll and write their shapes as chord books do (G is `320003`; see [The music: how a guitarist plays a chord](#)). Why does the F shape have no open strings? Which shape would you play for E7, and which notes of the chord does it double? Check your answers with `open_chords_are_the_ones_in_chord_books` and `other_keys_take_barre_chords`.

24. Full chords through distortion (lineup-core, easy)

In `band/extras.rs`, change Rock beat's guitar from

`GuitarPart::Power(Comping::Pulse)` to `GuitarPart::Strum(Comping::Pulse)` : full shapes through the overdriven guitar. Listen with a SoundFont. Why do rock guitarists avoid this? Put it back.

25. A guitar for Lo-fi (lineup-core, easy)

Lo-fi has no extra players. Give it a nylon-string guitar picking the chords,

`GuitarPart::Strum(Comping::ArpUp)`, in `BY_STYLE` of `extras.rs`. Then misspell the style's name: which test fails, and what does it print?

Practises: tables of constants, tests that guard names.

26. Think about it: the guitar

- Over a major chord the boogie shuffle rocks between the fifth and the sixth; over a minor chord it stays on the fifth (see `the_boogie_alternates_fifth_and_sixth`). What chord do A, C, E and F make, and why would a minor sixth over A minor be risky? What would a major sixth, F#, turn the chord into?
- `try_frets` takes `found: &mut impl FnMut(&[u8])`. Try `found: impl FnMut(&[u8])`. What does the compiler say about the recursive call, and why?

27. A second keyboard (lineup-core, harder)

Add an electric piano (program 4) playing the style's comping an octave up in the **R&B** and **Neo soul** styles. Which parts of `band.rs` and `file.rs` must change (look at `written` and `written_too` in [In the file](#))? Check that a song whose band has only three players saves with the same format number as before.

28. Horns held and struck (the app, easy)

Tick **Horn section** in **Swing** and in **Pop ballad**, over the same chords. Compare the horns' notes in the piano roll: what is the same, and what differs? Why does smooth voice leading matter even more for long notes than for short ones?

29. Common tones (the app, easy)

Type `C Am F G` in **Pop beat** with the horns, and write down each chord's horn voicing from the piano roll. Which horn keeps its note from one chord to the next, and for how many chords? A note two chords share is a **common tone**: why does the horn on it not move?

30. Cheap fifths (lineup-core, easy)

`cost` charges `PARALLEL_FIFTHS`, 24 semitones, for each pair of horns moving in parallel fifths. Make it 2 and run `cargo test -p lineup-core horns`. Which test fails, between which two chords, and which horns move in fifths? Put it back.

31. Think about it: names in two tables

`HELD` and `BY_STYLE` in `horns.rs` name styles by their names. What happens if a style is renamed in the library? Find the test that would catch it.

32. A split point (lineup-app, medium)

Arranger keyboards often have a **split point**: only the keys below it choose the chord, so the right hand can play a tune. Add one, say below C4, as a second option next to **Accompaniment follows chords**. Which keys reach `chord_held` now, and what still happens to the keys above?

33. Two keys a major second apart (lineup-core, easy)

`chord_held` reads two keys by the interval between them (see [Following the chords you hold](#)). Make two keys a major second apart a `sus2` chord on the lower one. Which test needs a new case? Before your change, holding F below G asked for no chord; what does it ask for now, and is that what a player holding them would mean?

34. Think about it: chords at once

Notes already sounding keep their old pitch until they end, so a chord held for a whole bar changes only at the next bar. Arranger keyboards play the new chord at once. What would the audio thread need to know to do that, and what would it cost?

The orchestra

35. Clap with the bass drum (the app, easy)

Apply the Orchestra to **The Rite of Spring, The Augurs of Spring** and mute the **Song** track. Clap along with the bass drum. Can you predict where the next stroke comes? Unmute the song and listen to the horns' accents falling on the same eighths.

36. Felt beats by hand (pencil, easy)

`TimeSignature::felt_groups` gives 13/16 as $2 + 2 + 2 + 2 + 2 + 3$. On which sixteenths do the horns strike in a bar of 13/16 where they keep to the felt beats? Open **Petrushka, The Moor's Room**, apply the Orchestra, find a bar of 13/16 on the ruler and check. If they play something else there, which rule of [Playing along with the score](#) explains it?

37. in_meter for the tresillo bar (lineup-core, easy)

Work out by hand the line `in_meter` writes for a bar of $3+3+2/8$ from the Orchestra's `x... .... o... ....`. Then add it to `a_rhythm_that_follows_the_meter_plays_each_bar_in_its_own` in `rhythm.rs`, and run it.

38. A style in 4/4 over changing meters (the app, easy)

With the Orchestra playing the **Ritual of Abduction**, click **Pop beat** in the Library. What happens to the ruler, and why does a style in 4/4 have to do that, when the Orchestra did not?

39. Only the voices (the app, easy)

Open **Shchedryk (Carol of the Bells)**, apply the Orchestra with only **Voices / instruments** ticked, and open the **Mixer**. Which instruments do the Song track's channels play? Which do the new Piano and Bass tracks play?

40. Accents nobody wrote (lineup-core, easy)

In `band/following.rs`, change `TUTTI` from `(9, 5)` to `(3, 2)` and run `cargo test -p lineup-core accents_of_the_augurs`. Do accents appear that Stravinsky did not write? Put it back.

Form, variations, sections and locks

41. The intro leads somewhere (the app, easy)

Write `| Dm7 | G7 | Cmaj7 | Cmaj7 |`, a ii-V-I in C, and give it a 2-bar intro. Which chords does the intro write? They do not lead into C: what do they lead into, and why?

42. The last chord (the app, easy)

Set the key to A minor (the **Key** menu beside the harmony keys), write a song that starts on Am, and turn the ending on. Which chord does it end on? Now change the first chord to F: which chord does it end on now, and why is that still right?

43. How long is the song (the app, easy)

Export a song with a 4-bar intro, three choruses and an ending as a WAV file. How many bars long is it? Check your answer against the file's length and the tempo.

44. The form moves (the app, easy)

- Give a song an 8-bar intro, change a chord in it, then change the intro to 2 bars. What happens to your chord? Undo it.
- Type a chord in the first bar of the ending. Where is the ending now, and where is the end-repeat sign?

45. Think about it: the form in Rust

- `Form` is `Copy`; `Band` is not. Look at their fields: why could `Band` not be?

- In `played_pieces`, the chorus repeats only `if from <= start && end <= to`. What would a loop region from the middle of the intro to the middle of the chorus play without that test?

46. In two, then walking (the app, easy)

Choose the **Jazz ballad** style, write `| Cmaj7 | Am7 | Dm7 | G7 |` twice, and set bars 5 to 8 to variation b. Open the bass's patterns in the piano roll. How many notes does a bar have in a, and in b? Which notes does the bass play in the Dm7 bar of each?

47. Fills and crashes (the app, easy)

- In an eight-bar **Pop beat**, set bar 5 to variation b and bar 6 back to a. Which bars get a drum fill, and where do the crashes fall? Why could that sound restless in a real song?
- Choose **Swing** and play a few bars in each variation. What changes and what stays? Why does the time stay on the same cymbal?

48. A style for some bars (the app, easy)

- Write `| C | Am | F | G |` twice in **Pop beat**, and make bars 5 to 8 **Bossa nova**. Where does the bossa nova's bass line start, and how does it differ from the pop one?
- Play a section in **Chorale**, which has no drums. What do the drums play there, and in the bar before it?

49. Locks (the app, easy)

- Open **Ode to Joy**, lock **Keys** in the rack, and apply **Doo-wop**. Which tracks changed, and what did Keys keep?
- In a new song, lock the kick and apply **Bossa nova**. What does the kick play now?

50. Think about it: markers and borrows

- `tidy` for variation markers sorts, then calls `dedup_by`, then `retain`. What would go wrong if it called `retain` before sorting?

- `played_in_runs` takes `play: impl Fn(...)`. Could it take an `FnMut`? What would change where it is called?
- `where_it_plays` returns `impl Iterator`. What would change if it returned a `Vec`, and why does `notes.extend(...)` accept both?
- `StylePattern` holds a `&'a mut Pattern`. What does the borrow checker say if you call another `Project` method while one is alive, and why is that a good thing?

Stops and the soloist

51. Stop-time (the app, easy)

Type a twelve-bar blues in F with shots on the first four bars: `| F7. | F7. | F7. | F7. | Bb7 | Bb7 | F7 | F7 | C7 | Bb7 | F7 | C7 |`, and pick **Shuffle**. Play along in the gaps with the F blues scale (F A $\flat$ B $\flat$ B C E $\flat$). What changes in the feel when the band comes back in bar 5?

52. Pushes (the app, easy)

Type `| C | Am | F | G |` in **Pop beat** and listen. Then push every chord after the first: `| C | ^Am | ^F | ^G |`. Clap the piano's rhythm both ways. Which beat is no longer struck? Then compare the piano and the bass in the piano roll: how long does each hold a pushed chord?

53. A hold over two bars (lineup-core, medium)

Type `C..` in bar 1, nothing in bar 2 and `G7` in bar 3, so the hold lasts two bars, and look at the bass: one note. [Each player strikes its own way](#) says why the bass uses `bass_struck` and not `Bass::Held`. Write a test in `band.rs` that would fail if it did use `Bass::Held`.

54. A shot a quarter long (lineup-core, easy)

A shot is an eighth note: `SHOT_STEPS` in `band/stops.rs` is 2. Make it a quarter note and run `cargo test -p lineup-core`. Tests fail not only in `stops.rs` (`a_shot_strikes_once_and_the_band_stops`) but in the library's charts and demos

too: why do those care how long a shot is? Fix the expectations, or decide the change was wrong.

55. A push of a quarter (lineup-core, medium)

Add `^^^C`, a push of a quarter note: a variant of `Push`. Let the compiler show you every `match` that must learn it (`steps`, `marks`, `describe`). Which other places need it? (Hint: `read_marks` refuses three carets today, and `breaks_menu` in the app lists the pushes.)

56. Think about it: a fill before a stop

The drums play a fill in the bar before a change. Should a shot or a rest in that bar keep it? Find where `drum_part` adds the fill and where `apply_to_drums` takes the groove out, decide, and write a test for what you chose.

57. Guide tones and chord scales (the app, easy)

- With the soloist on, open its part in the piano roll and find the notes marked `3` and `7`. Where does a 7th move to a 3rd a half step away at a chord change? Sing only the 3rds and 7ths, one a bar, over the band.
- Type `| Am7 | D7 | Gmaj7 | E7 | Am7 | D7 | G6 | G6 |` in G major. Work out the scale of each chord with the rules of [Chord scales](#) before you look. One chord plays an F natural, although the key has F#: which one, and which other note of its scale is not in G major?
- In A minor, type `| Bm7b5 | E7 | Am | Am |`. Which note of E7's scale is not in A natural minor? Listen for it in the solo.

58. Leaps and enclosures (lineup-core, medium)

- The soloist leaps to the second chord tone one time in five (`dice.chance(20)` in `target`). Make it 60 and play a few solos with **Another solo**. How does the line change? Which tests still pass, and why?
- Add an **enclosure**, a favourite bebop figure: before a target, a scale note above it and then the note a half step below it, as D B C into C. Give it a new `NoteRole`, let the compiler show you every `match` that must learn it, and write a test in the style of `roles_name_what_each_note_does`.

59. A new solo every chorus (lineup-core, harder)

With **Choruses** set to 3, the soloist plays the same solo three times. Make each chorus a new solo. Where is the chorus repeated (look for `repeated_chorus`), and how could each time round get its own seed while the song still plays the same every time it is opened?

Your own styles and the practice tools

60. Read the files (the app and a text editor, easy)

Save a style from the style editor (see [Trying it](#)) and open the `.lineup-style` file in a text editor. Find its `format`, `bars`, `chart` and the `lanes` with each channel and its notes. Then apply it to a song, save the song, and search its file for `custom_rhythm`: the whole style is inside. Why must it be, and what is the song's format number at least?

61. Out-of-key dominants (lineup-core, medium)

Over a chord outside the key, `adapt_pitch` in `user_style.rs` keeps the distances written over C, so the E7 that leads to Am plays E F# G# A B C# D (see [Chords from outside the key](#)). The soloist's chord scales, in `harmony/chord_scale.rs`, would give it the F and C of A minor. Use them for dominant chords outside the key. Which written notes of a style over E7 in C major now play differently? Write the test first, in the style of `template_notes_stay_in_the_key`, and listen: is a pattern that sounds the same over every chord worth more than the right notes?

62. How many bars (pencil, easy)

A user style holds at most 256 steps and at most sixteen bars. How many bars can a style in 7/8 have, and one in 6/4?

63. The metronome counts (the app, easy)

- Turn on the metronome over **Silent Night**, in 6/8. How many clicks are there in a bar, and where does the high one fall? Now **Greensleeves**, in 3/4: the same six

eight notes to a bar, counted differently.

- With the count-in on, pause a song halfway through a bar, then press Play again. What do you hear before the music, and why is that better than counting a whole bar and then starting halfway through the next?
- Set the speed to 50% and export a WAV file. How long is it, compared with the song at full speed? Why?

64. Tapping (the app, easy)

- Tap 4 times at an even pace, then 4 more times faster without stopping. Which tempo do you get, and why not the average of the eight gaps?
- In a song in 6/8, tap 80 times a minute. Which tempo does the song get, and why?
- Take over the bass with **You play**, then change a chord, and give the bass back: is the new chord in its part?

65. Think about it: the click and the count-in

- `ClickVoice::strike(Click::None)` returns at once. What would you hear if it set `level` to 0 instead?
- In `begin_play`, what goes wrong if `self.stop()` is not called when Play arrives while the song is already playing?

66. The speed trainer by hand (the app, easy)

- Set **from 50% to 100% by 10% every 2 passes**. Write down the speed of each of the first eight times round, then play it with the trace on and check.
- Set **from 100% to 60%**. What does the trainer do now? Think of a use for it (hint: a passage you can play fast, but not yet evenly).
- Start a run, Pause in the middle of it, then press Play. Then Stop and press Play. What is the speed each time, and why is it useful that the two buttons differ?
- With the trainer on and nothing looped, play a whole song. When does the speed go up?

67. Think about it: the trainer's messages

- In `came_round`, `self.laps_seen.replace(laps)` runs before the check for `self.training`. What would go wrong when you turn the trainer on in the

middle of a song if it ran after it?

- The audio thread throws an old list of lap tempos away with `throw_away` when a new request replaces it. What would happen if it simply assigned the new list, and why does that matter on the audio thread in particular?

68. More for the ear (lineup-core, medium)

- **A tune for the major seventh.** `Interval::tune` has none going up. Find a song whose melody starts with one, check it on a piano, and add it. Does `going_up_names_a_tune` need to change?
- **Tunes going down.** Add a `tune_down` (the start of Beethoven's Fifth Symphony is a major third down) and use it when the question went down.
- **What you get wrong.** Keep, for each interval, how often it was asked and how often it was right, and ask the weak ones more often. How would `IntervalQuestion::new` use the counts without losing the rule "same seed, same question"?

69. Think about it: ear training

- Sixth chords are left out because C6 and Am7 have the same notes. Could they be asked fairly if the chord always played over its root an octave below? Would inversions still make sense?
- Let the progression exercise play the song's own chord sheet, with the names hidden. What would it need from `chord_sheet.rs`?

The lead sheet and parts

70. Notation choices (lineup-core, medium)

- **Courtesy accidentals.** When a bar has an F# and the next bar an F natural by the key signature, many lead sheets print a natural in brackets as a reminder. Where would `BarAccidentals` need to know about the bar before?
- **Half-diminished.** Some lead sheets write Em7 $\flat$ 5 as E $\emptyset$ 7. Add a choice for it. Is it a choice of `layout.rs`, or of the chord's name?

71. Think about it: the page

- **Endings.** When the last bar of a repeated section differs the second time, lead sheets use first and second endings: a bracket "1." over one bar, "2." over the other. What would the playlist need to tell `score.rs`?
- **Bars of rest.** A tune silent for eight bars could show one bar with a thick line and "8" over it. Where in `systems` would it go, and how does it change the bar numbers of the next line?

72. More instruments (lineup-core, medium)

- **The exact octave.** Add a choice that writes every part exactly as the instrument plays it: a ninth up for the tenor saxophone, an octave and a sixth for the baritone. Which function changes? How many ledger lines does a tune around C5 need for the tenor, and why do books of lead sheets not do this?
- **D and G instruments.** A trumpet in D sounds a whole tone higher than written, an alto flute in G a fourth lower. Add them. What do `note()` and `semitones()` return, and what else in the window and the label has to know about them?
- **The Eb clarinet** sounds a minor third *higher* than it reads. Is that a new `Transposition`, or the alto saxophone's part in another octave? What would `shift` need to know to tell them apart?

73. Parts as text, and the concert key (lineup-core, easy)

- Give **Export chord sheet...** the same choice of instruments. Which single function does the work for every chord? (See [The chords: moved letters and all.](#))
- Some parts print the song's own key under the instrument, "Concert key: B $\flat$ ". Where in `layout.rs` would it go, and what does `lay_out` need to know that it doesn't now?

How a demo is written

Files: `crates/lineup-core/src/library/song.rs`, `crates/lineup-core/src/library/song/`, `crates/lineup-core/src/library/chart.rs`, `crates/lineup-core/src/library/demo.rs`, `crates/lineup-core/src/library/interpret.rs` and `crates/lineup-app/src/app/demo_menu.rs`

File > Demos opens 343 songs, from nursery rhymes to the Rite of Spring, and a short demo of every style, twelve-tone style and micromode style. Click one and it opens, its chords on the chord sheet, and plays. What to listen for in them is in the manual's [listening guide](#). This chapter is about what a demo *is*.

A song demo is not a file. It is a Rust constant, compiled into the program: the tune and the chords written as text, the way a lead sheet writes them, and a few fields that say how the song is played. `song_demo` turns the constant into an ordinary `Project` each time you open it. That has three good sides:

- **The tests read every song.** A bar one sixteenth too long, a chord that cannot be read, a form that names a part that does not exist: each fails `cargo test` before anyone hears it.
- **A demo is the same everywhere.** It cannot be missing, damaged or out of date; a new version of the app brings its own copy.
- **What the songs need, the library learns.** A piece that changes key, meter or tempo, a round, an orchestra handing its tune from the flute to the clarinet: each became a feature of `Part`, and every song can use it.

A demo is read-only for the same reason: there is no file to write your changes back to, so **Save** asks for a new one (see [The title bar](#)).

A song is a constant

Here is a whole song, the round Frère Jacques:

```

pub(super) const FRERE_JACQUES: Song = Song {
  name: "Frère Jacques",
  origin: "French round, 18th century",
  feel: "Round",
  genre: Genre::Folk,
  tonic: C,
  mode: Major,
  time: COMMON,
  tempo: 104.0,
  comping: Comping::ArpUpDown,
  bass: Bass::RootFifth,
  rhythm: "Folk",
  band: SONG_AND_GUITAR,
  style: "Folk",
  parts: &[
    Part::new("Alone", FRERE_JACQUES_TUNE, FRERE_JACQUES_CHORDS),
    Part::new("Round", FRERE_JACQUES_TUNE,
FRERE_JACQUES_CHORDS).with_voices(&[
      (1, 0),
      (2, 2),
      (3, 4),
      (4, 6),
    ]),
    // ...
  ],
  form: &["Alone", "Round", "Round again", "Last voices"],
  drums_from: 0,
  about: "A round: four voices sing the same tune, each two bars after
the last. It works \
      because every two bars of the tune fit the same chord.",
};

```

The tune and the chords are constants of their own above it, because four parts use them:

```

const FRERE_JACQUES_TUNE: &str = "C5:4 D5 E5 C5 | C5 D5 E5 C5 | E5 F5 G5:8
| E5:4 F5 G5:8 | \
      G5:2 A5 G5 F5 E5:4 C5 | G5:2 A5 G5 F5 E5:4 C5 | C5 G4 C5:8 | C5:4 G4
C5:8";
const FRERE_JACQUES_CHORDS: &str = "I | I | I | I | I | I | I | I";

```

The fields read top to bottom like the heading of a lead sheet: what it is and where it comes from, its key, meter and tempo; then how the band dresses it, a way of comping and a bass line from the library (see [Comping: how the chords are played](#)), a rhythm by name, the instruments (`band`) and the style it is closest to, which the Library shows selected while the demo is open. Then the music: the **parts**, each a section such as a verse or a chorus, and the **form**, the order they are played in, by name. A part played three times is written once and named three times. `drums_from`

is the entry of the form where the drums come in, 0 for the start, 1 when the song opens with a pickup or an introduction the drums leave alone.

One file per genre

343 songs would make one file of tens of thousands of lines. So `song.rs` holds the types, the shared constants, the list and the tests, and the songs live next to it, one file per genre:

```
library/  
  song.rs          Song, Part, Band, SONGS, the tests  
  song/  
    balkan.rs  
    blues.rs  
    classical.rs  
    ...  
    tango_played.rs
```

`mod tango;` inside `song.rs` makes Rust look for `song/tango.rs`: each genre file is a **child module** of `song`. A song is `pub(super) const`, visible to its parent and no further: nothing outside the library needs the name `HABANERA`, only the song, which it finds in the list.

Each genre file starts with one line:

```
use super::*;
```

`super` is the parent module and `*` brings in every name it can see, its private ones too: in Rust an item private to a module is visible to that module's children. So a song can use `COMMON`, `SIX_EIGHT`, the bands and every other constant `song.rs` keeps to itself. Two of the parent's own `use` lines matter most:

```
use PitchClass::*;  
use ScaleKind::*;
```

They let a song write `tonic: A` and `mode: Dorian` rather than `PitchClass::A` and `ScaleKind::Dorian`. Genre is not imported that way, on purpose: `Genre::Blues` and `ScaleKind::Blues` would both be `Blues`, and two glob imports with the same name are an error as soon as the name is used. So a genre is always written in full, `genre: Genre::Blues`, and `mode: Blues` is the scale.

The list

```
pub static SONGS: &[Song] = &[
    pop::AULD_LANG_SYNE,
    pop::LONDON_BRIDGE,
    pop::OLD_MACDONALD,
```

Every song is listed by name. Rust cannot join slices together while it compiles, so the genre files cannot each bring a list of their own; naming each `const` is the plain way. A `const` is copied in wherever it is used, so the list holds the songs themselves, and being a `static`, it has one place in memory for the whole run: the menu and the Library can hand out `&'static Song` references into it. The list is grouped by genre, in the order of `Genre::ALL`, and a test checks it with `windows(2)`, which gives every pair of neighbours:

```
SONGS.windows(2).all(|pair| pair[0].genre <= pair[1].genre),
```

`<=` works on genres because `Genre` derives `PartialOrd`, which orders an enum's variants as they are written. The menu sorts the songs by title on its own (see [The Demos menu](#)), so a new song goes at the end of its genre's block, and two changes adding songs at once do not fight over the same lines.

Where the notes come from

Each song carries `origin`, who wrote it and when, and a `// Source:` comment above it naming the written score, lead sheet, engraving or recording its notes were taken from, and what was simplified. The Promenade's comment, for instance, names the LilyPond engraving it was read from and explains how a chord was chosen for each beat. Almost every song is old enough to belong to everyone; the few that are still under copyright somewhere say so in their comment. Twenty-five are new tunes for genres with few free ones, written the way those genres write theirs, and their `origin` says "Written for Ferrous Lineup". How a performance or a score becomes a constant is the next chapter, [From a MIDI performance or a score to a demo](#).

The tune, as text

A melody is written in a small notation, read by `chart::parse_melody`:

Text	Means
<code>C4 , F#5 , Bb3</code>	a note: its letter, <code>#</code> or <code>b</code> , and its octave (<code>c4</code> is middle C)
<code>:4</code>	its length in steps (4 is a quarter note); without it, as long as the note before
<code>:1/2 , :1/4</code>	shorter than a step: a thirty-second, a sixty-fourth, for grace notes and runs
<code>-:8</code>	a rest, here half a bar of 4/4
<code>_:8</code>	a tie: the note before goes on, across a bar line too
<code> </code>	a bar line: it changes nothing, but each bar is checked
<code>A3+C#4+E4:4</code>	a chord, its notes struck together
<code>&</code>	another voice, from the start, at the same time: <code>E5:4 D5 C5 B4 & C3:16</code>
<code>(3</code>	the next three notes or rests are a triplet, three in the time of two
<code>!p , !f , !sfz</code>	a dynamic, from <code>!pp</code> to <code>!ff</code> , or a sforzando on the next note
<code>!< , !></code>	a crescendo or a diminuendo, up to the next dynamic
<code>>A4:2 , D5:2.</code>	an accent before a note; a staccato dot after it

The marks take no time. A triplet's notes start between the steps of the grid, so they are placed to the tick (see [Notes between steps](#)). The verse of Greensleeves reads like this:

```
C5:8 D5:4 | E5:6 F#5:2 E5:4 | D5:8 B4:4 | G4:6 A4:2 B4:4 | ...
```

A tune that begins before its first bar, as the "In a" of "Clementine", has a **pickup**. It is written as a part of its own, a bar that is nearly all rest:

```
Part::new("Pickup", "-:8 C5:3 C5:1", "I"),
```

That is one bar of 3/4: two beats of rest, then "In a", a dotted eighth and a sixteenth. The first strong beat of the tune then falls on the first beat of the verse, where the drums and the chords expect it.

The chords, as a chart

Under the tune goes a chord chart, bar by bar (see [Chord charts: chords by the bar](#)): numerals such as `i | bIII | bVII | v`, or names as a lead sheet writes them, `C#m | G#7/B#`, read in the part's key. The marks of the chord sheet work too: a chart can push a chord or stop the band (see [Shots, holds, rests and pushes](#)). The chart is what the band plays, with the song's comping and bass line, and what the chord sheet shows.

Some music's accompaniment is part of the piece: a rag's left hand, a Bach chorale, a choir. Then the part writes it out, and its chords are not accompanied:

- `with_left(...)` writes notes for the **Keys** channel, in the notation of tunes, `&` and all;
- `with_lines(&[...])` writes **other instruments**, each a `Line` with the name of its channel, a General MIDI program and its notes. Lines with the same name in any part play on the same channel, so the Misa Criolla's choir is one line, "Choir", with its alto, tenor and bass joined by `&`.

A written-out part may still have a chart: then it names what the music plays, for the chord sheet, and the band leaves it alone.

Parts that change something

`Part::new(name, melody, chords)` makes a plain part. The others are the same part with one field changed, by a `const fn` that uses **struct update syntax**:

```
const fn in_key(self, tonic: PitchClass, mode: ScaleKind) -> Part {
    Part {
        key: Some((tonic, mode)),
        ..self
    }
}
```

`..self` means "every other field as in `self`". Because it only moves values around, it can be `const`, and a song stays a constant with `.in_key(FSharp, Major)` at the end of one of its parts. The rest work the same way:

Method	The part...	For example
<code>in_key(tonic, mode)</code>	is in another key: a modulation	the Mountain King climbs to F# major

Method	The part...	For example
<code>in_time(time)</code>	is in another time signature	a refrain in 3/4 in a song in 2/4
<code>in_meters(&[...])</code>	has a time signature for each bar	the Promenade's 5/4 and 6/4
<code>with_tempo(&[...])</code>	has the score's tempo markings	"Più vivo", a ritardando, a fermata
<code>played(comping, bass, rhythm)</code>	is accompanied its own way	the refrain in 3/4 needs a waltz's comping
<code>with_voices(&[...])</code>	is sung by several voices	a round
<code>played_by(&[...])</code>	hands its tune to other instruments	Boléro, from the flute to the clarinet
<code>marked(dynamic)</code>	is played this loud	Boléro's pianissimo start
<code>with_drums(&[...])</code>	has its drums written out	Boléro's snare drum
<code>with_left, with_lines</code>	writes out its accompaniment	a rag's left hand, a choir

In the Hall of the Mountain King uses two of them:

```
Part::new("Higher", MOUNTAIN_HIGHER,
MOUNTAIN_HIGHER_CHORDS).in_key(FSharp, Major),
// "Più vivo": faster at once, at Grieg's own metronome mark.
Part::new("Più vivo", MOUNTAIN_THEME, MOUNTAIN_CHORDS).with_tempo(&
[tempo(0, 154)]),
```

A key of its own

A part in another key reads its numerals in that key, so **I** in the F# major part is an F# chord, and the demo puts a key marker where it starts and another where the music goes back. The song answers which key a part is in:

```
pub fn part_scale(&self, part: &Part) -> Scale {
    match part.key {
        Some((tonic, mode)) => written_as_tune(Scale::new(tonic, mode),
[part]),
        None => self.scale(),
    }
}
```

`written_as_tune` decides the key's *name*. A `PitchClass` has one variant for each of the twelve notes, with sharps (`ASharp`, never a B flat), but the Promenade is in B $\flat$ major, not A $\sharp$ major, and Take Five in E $\flat$ minor, not D $\sharp$ minor. Rather than make every song spell its key, the library counts the accidentals of the tune and names the key the way the tune is written:

```
fn written_as_tune<'a>(key: Scale, parts: impl IntoIterator<Item = &'a
Part>) -> Scale {
    let (sharps, flats) = parts
        .into_iter()
        .map(|part| accidentals(part.melody))
        .fold((0, 0), |(sharps, flats), (s, f)| (sharps + s, flats + f));
    match sharps.cmp(&flats) {
        Ordering::Greater => key.with_spelling(Spelling::Sharps),
        Ordering::Less => key.with_spelling(Spelling::Flats),
        Ordering::Equal => key,
    }
}
```

`fold` carries a pair through the parts, adding up both counts at once, and `cmp` returns an `Ordering`, which the `match` reads as “more sharps”, “more flats” or “as many”. A tune with as many of each, or none, leaves the key its usual name. How a key is spelled is in [Sharps or flats: how a key is written](#)

Music in no key at all, where all twelve notes are at home, says so with the chromatic scale: the Ritual of Abduction from start to end, with `mode: Chromatic`, and parts of Petrushka with `.in_key(C, Chromatic)`. Such a part has no key signature, and every note it plays counts as in its key.

A meter for each bar

Most parts are in the song's time signature or one of their own. Some music changes meter nearly every bar: the Promenade alternates 5/4 and 6/4, the Rite of Spring goes through 2/8, 3/8, 3/4, 5/8 and more. A part with `in_meters(&[...])` has one time signature per bar. A pattern has one time signature, so `Song::part_runs` cuts such a part into **runs** of bars in one meter, each a part of its own with its share of the tune and the chords:


```

let mut runs: Vec<(usize, usize, TimeSignature)> = Vec::new();
for (bar, &time) in part.meters.iter().enumerate() {
    match runs.last_mut() {
        Some((_, bars, last)) if *last == time => *bars += 1,
        _ => runs.push((bar, 1, time)),
    }
}

```

`last_mut` gives the last run, if there is one, to change in place: a bar in the same meter makes it one bar longer, and any other bar starts a new run. The pieces of the text come from `bars_of`, which returns a slice of the part's own `&'static str` between two bar lines, so a run is a `Part` like any other and costs no copy. The demo names the pattern of each later run after the bar it starts on, "Call, bar 2", and puts a time marker wherever the meter changes. The Rite of Spring would need more patterns than a project allows, so a run whose notes are the same as an earlier run's, in the same meter, plays that earlier pattern again, as a producer reuses a loop.

The score's tempo

A `TempoMark` is one of three things a score writes:

```

pub enum TempoMark {
    /// From `at` on, `bpm` beats a minute, at once: a new section with
    /// a new tempo, such as "Più mosso" (more movement), "Adagio" or
    /// "Tempo I" (the first tempo again).
    At { at: usize, bpm: u16 },
    /// From `at`, slowing down or speeding up bit by bit until `bpm`,
    /// reached `over` steps later: "rit." or "rall." (slowing down),
    /// "accel." or "stringendo" (speeding up).
    Glide { at: usize, bpm: u16, over: usize },
    /// A fermata (⤿), the "bird's eye" over a note or a rest: from `at`,
    /// for `steps` steps, the music waits, lasting twice as long; then
    /// it carries on at the tempo before.
    Fermata { at: usize, steps: usize },
}

```

`tempo(at, bpm)`, `glide(at, bpm, over)` and `fermata(at, steps)` are `const fn`s that build them, counting steps from the start of the part. The demo turns each into tempo markers on the playlist, every time the part is played. A fermata becomes two: half the tempo for as long as the held note lasts, then the tempo before. The project has no fermata of its own; half as fast is exactly "twice as long" (see [Tempo changes](#)).

Rounds and canons

`with_voices` says who sings the tune, and from which bar of the part: one (voice, bar) for each copy. Frère Jacques' round is four voices, each two bars after the one before:

```
Part::new("Round", FRERE_JACQUES_TUNE, FRERE_JACQUES_CHORDS).with_voices(&[
    (1, 0),
    (2, 2),
    (3, 4),
    (4, 6),
]),
```

A negative bar is a voice that began in the part before, so only its end is heard: "Round again" lets voices 2 to 4 finish their last bars while they start again. Voice 1 is the Lead channel, and every other voice gets a channel of its own, "Voice 2", "Voice 3" and so on.

Other instruments, louder and softer

`played_by` takes a list of (program, semitones) : each instrument plays the tune, that many semitones higher, on a channel named after it. Boléro hands its tune from the flute to the clarinet, the bassoon and on; its Eb clarinet plays it an octave up, (gm::CLARINET, 12). `marked` sets how loud the part is played, from pianissimo to fortissimo:

```
pub fn scale(self) -> f32 {
    match self {
        Dynamic::Pianissimo => 0.4,
        Dynamic::Piano => 0.55,
        Dynamic::MezzoPiano => 0.75,
        Dynamic::MezzoForte => 1.0,
        Dynamic::Forte => 1.15,
        Dynamic::Fortissimo => 1.3,
    }
}
```

`with_drums` writes a drum part out, in the notation of tunes, with `x` for a stroke and `X` for an accented one. Rather than a second parser, `drum_line` turns each `x` into a C4 and each `X` into a D4, reads the line as a melody, then makes every note a one-step hit, accented if it was a D4, as loud as the line's dynamics say: triplets, rests and dynamics come for free.

The band of a song

`band` names the General MIDI programs of the Lead, the Keys and the Bass channels, and maybe a horn section:

```
pub struct Band {  
    pub lead: u8,  
    pub keys: u8,  
    pub bass: u8,  
    pub horns: Option<Horns>,  
}
```

Most bands are constants shared by the songs of a genre:

```
const BOSSA: Band = Band::new(gm::TENOR_SAX, gm::NYLON_GUITAR,  
gm::ACOUSTIC_BASS);
```

`Band::new` and `with_horns` are `const fn s`, so a band can be built inside a constant. A horn section is a program and a way of comping: a big band's saxophones hold the chords (`Comping::Held`) and its brass punches them on the beat and just before beat 3 (`Comping::Charleston`). A tango orchestra's violins are a "horn section" too, with a string program: the demo calls the channel "Strings" for programs 40 to 51 and "Horns" for anything else. A song with a band of horns is a different thing from the band style's [horn section](#), which any song can switch on.

From a song to a project

`song_demo(song)` builds the project, in four steps.

Channels. The Lead, a channel for each extra voice, the Keys only if some part plays chords or a left hand, the Bass unless the song has no bass line, the horns, a channel for each instrument a tune is handed to, and one for each line's name. Each gets its program.

A pattern for each part, or for each run of a part whose meter changes. Into it go the tune on its voices' channels, the chart accompanied with the part's comping and bass (`accompany_in_bars` , which keeps the comping in step with the bars, so a chord a beat long is still struck), the horns' part, the written-out left hand and lines, and the written-out drums. Then two passes make them sound played (see the next section) and the part's dynamic scales them.

A drum pattern for each part, from the song's rhythm, or the part's own, in the part's meter, with a fill at the end of parts four bars or longer, unless the part writes its drums out.

The arrangement. The form is walked once, each entry placing its part's patterns one after another on the Song track and its drums on the Drums track from `drums_from` on, with key, tempo and time markers where they change and the chart written on the chord sheet. A last bar on the home chord, with a crash, ends the song, in the key of the last part, so a song that modulates ends in the new key. A piece whose last part is written out ends where its score ends, and so does a chart that ends on a break, as a tango's "chan-chan": the score already says how the music stops.

Two small touches finish it. Songs, hymns, tangos and concert pieces slow down a little in their very last bar, as a player does where there is no drum machine to keep up with:

```
fn ends_broadening(song: &Song) -> bool {
    use super::Genre::*;
    matches!(song.genre, Classical | Folk | Soul | Tango | Folklore)
}
```

That is a *poco ritardando* to nine tenths of the tempo, unless the score already marks how the last part's tempo goes. And the swing is the song's rhythm's, except in folk and classical music, whose eighths stay even over any beat:

```
pub fn swing(&self) -> u8 {
    match self.genre {
        Genre::Folk | Genre::Classical => 0,
        _ => Rhythm::named(self.rhythm).map_or(0, |rhythm| rhythm.swing),
    }
}
```

The result is an ordinary project, to take apart and change: nothing in it is marked as a demo.

Playing like a person

A score written out note by note, every note as hard as the next, sounds like a machine. `interpret.rs` changes the velocities, a few steps out of 127, the way a player would; the notes stay where they are. For a tune, `perform` shapes each phrase:

- the first beat of a bar a little harder, the other beats less, the notes off the beat lighter;

- the higher a note in its phrase, the more it is sung out;
- a long note leaned on, the last note of a phrase relaxed;
- and no two notes quite alike.

A phrase ends at a rest of an eighth note or more, the shortest breath a singer takes, or after four bars. The chords and the bass, whose accents the comping already wrote, only lose their evenness: the notes of a chord a little apart from each other, the chords a little more. Then `follow` makes the band as loud as the tune is marked where each of its notes starts, so a crescendo in the melody carries the band with it, and an accent in the tune is half an accent in the band.

“No two quite alike” must not mean “different every time”: a demo has to sound the same each time it is opened, and a test that checks its notes must see the same notes. So the small differences come from a function that looks random but is not:

```
fn wobble(seed: u64, value: u64) -> f32 {
    let mut x = seed ^ value.wrapping_mul(0x9E37_79B9_7F4A_7C15);
    x = (x ^ (x >> 30)).wrapping_mul(0xBF58_476D_1CE4_E5B9);
    x = (x ^ (x >> 27)).wrapping_mul(0x94D0_49BB_1331_11EB);
    x ^= x >> 31;
    (x >> 40) as f32 / (1u64 << 23) as f32 - 1.0
}
```

This is the mixing step of **SplitMix64**: multiplications and shifts that scatter the bits of the input so thoroughly that neighbouring inputs give unrelated outputs. The input is the note’s step and pitch; the seed is made from the song’s, the part’s and the channel’s names with FNV-1a (see [FNV-1a](#)). The top 24 bits of the result, divided by 2^{23} , give a number from 0 to 2, and the minus one makes it run from -1 to 1.

`wrapping_mul` multiplies and throws away what does not fit in 64 bits, which plain `*` would treat as an overflow. The shaping is kept small next to the difference between piano and forte: a velocity changed by `clamp` stays between 24 and 124, or where it was written if that is further out, so a pianissimo stays soft.

The Demos menu

The menu lists the demos by kind (**Songs, Styles, Twelve-tone, Micromodes**), songs and styles by genre, and every list in alphabetical order. The library’s own lists keep the order they were written in; the menu sorts a copy, built the first time it is shown, in a `LazyLock`:

```
static DEMOS: LazyLock<Demos> = LazyLock::new(||
Language::English.with(all_demos));
static DEMOS_ES: LazyLock<Demos> = LazyLock::new(||
Language::Spanish.with(all_demos));
```

A `static` must be known when the program is compiled; a `LazyLock` is a `static` whose value is worked out by the closure the first time anyone reads it, and kept. There are two, because the titles are sorted as they are shown, and Spanish shows many in Spanish.

Alphabetical, for people

Comparing text letter by letter, by Unicode number, puts every capital before every small letter and “Ábalos” after “Zamba”. So the menu sorts by a **folded** copy of each title: small letters, no accents, no punctuation, a hyphen as a space. “Für Elise” sorts as “fur elise”, right after “Funeral March”:

```
fn by_title(a: &str, b: &str) -> Ordering {
    fold(a).cmp(&fold(b)).then_with(|| a.cmp(b))
}
```

`then_with` breaks a tie, two titles that fold the same, by the titles themselves, so the order never depends on chance. `fold` is one `match` over the characters: each accented letter becomes its plain one, `#` and `b` become `#` and `b`, letters, digits, `#`, `/` and `+` stay (so “F#”, “3/4” and “2+2+3/8” can be searched), and any run of spaces and marks between words becomes one space.

Search

A search box sits above the submenus and has the keyboard as soon as the menu opens. Each demo has three folded texts: its title (in both languages, so “Noche de paz” finds Silent Night in either), its details (who wrote it, its feel, genre, key and time signature) and the notes about it. Every word typed must start a word in one of them: `bach` finds Bach, not Offenbach:

```
fn starts_a_word(text: &str, words: &str) -> bool {
    text.match_indices(words)
        .any(|(at, _)| at == 0 || text[..at].ends_with(' '))
}
```

The matches are ranked, best first:

```
enum Rank {
    TitleStart,
    TitleWords,
    Details,
    Words,
    About,
}
```

The title starting with what was typed comes first, then every word found in the title, then the phrase found in the details, as “d minor” in a song in D minor, then the words found apart, and last a word found only in the notes. `Rank` derives `Ord`, so the variants compare in the order they are written, and a **stable** sort by rank keeps the alphabetical order among equals. The two weakest ranks are shown fainter, so the matches you most likely meant stand out.

Testing it

The songs are data, and the tests read all of it. In `song.rs`:

```
every_bar_of_every_part_adds_up (every bar of every tune and left hand is as long
as its time signature, every chart can be read, a part with a meter per bar has as many
meters as bars), forms_name_real_parts_and_rhythms_exist,
every_song_names_a_real_style, songs_in_three_play_in_three,
songs_that_change_meter_have_a_style_that_follows_it,
every_genre_has_songs_and_they_are_in_order,
keys_are_written_as_their_tunes_are (Take Five in E $\flat$  minor, the Maple Leaf Rag's
trio in D $\flat$  major), melodies_are_mostly_in_their_key (at least 80% of a song's notes
in its part's key, which catches an F written for an F $\sharp$ ) and
the_first_notes_are_right (Ode to Joy starts F $\sharp$  F $\sharp$  G A).
```

In `demo.rs`, `every_song_makes_a_demo` builds all 343, and others check one thing each: `a_round_has_its_voices_in_the_right_places`,
`a_part_in_its_own_time_gets_time_markers`,
`runs_of_the_same_bars_share_a_pattern`, `tempo_markings_go_on_the_playlist`,
`songs_and_hymns_slow_down_a_little_at_the_end`,
`a_song_that_ends_after_a_modulation_ends_in_the_new_key`,
`a_piece_written_out_in_full_ends_where_its_score_ends`,
`a_big_band_has_its_horns`,
`the_band_follows_the_dynamics_of_the_tune_and_the_breaks_of_the_chart` and
`every_demo_keeps_a_sensible_tempo` (every tempo marker between three tenths
and two and a half times the song's tempo). `interpret.rs` checks that
`piano_stays_soft_and_fortissimo_loud` and that

```
a_tune_gets_its_accents_but_stays_close_to_how_it_was_written . demo_menu.rs
```

checks the folding, the order and the ranks, such as

```
matches_only_in_the_notes_come_last .
```

Beyond the unit tests, any demo can be played headless with the [playback trace](#),
`ferrous-lineup --trace "Frère Jacques" --out round.tsv`, and `ferrous-lineup -`
`-demos` lists the names it takes.

From a MIDI performance or a score to a demo

Files: `crates/lineup-core/src/library/song/`, `crates/lineup-core/src/midi_file/import.rs` and `crates/lineup-core/tests/key_finding.rs`

A demo is a constant of text (see [How a demo is written](#)). Short tunes are typed in by hand. Longer pieces are read from a version a computer can read, a MIDI file, an engraving or a lead sheet, by small scripts written for each kind of source. The scripts are not part of the app: what they write is, as ordinary Rust constants, and the `// Source:` comment above each song names what it was read from and what was left out. This chapter is about the methods: how a beat, a tune, a key and chords are read from notes, how the notes get their names, and what a demo keeps of a score. Most of it is music theory put to work. What to listen for in the pieces is in the manual's [listening guide](#).

Kinds of source

Source	What it holds	For
ABC (the collections of John Chambers and Henrik Norbeck, The Traditional Tune Archive)	a tune written as letters, often with chord names	folk songs and dances, the Balkans
MusicXML lead sheets (OpenEWLD, the public-domain part of the Wikifonia archive)	a tune with the chord names over it	songs of most genres: swing standards, blues, Latin, soul, pop
LilyPond engravings (the Mutoxia Project, OpenScore)	the whole score, every voice, as text	classical pieces
MIDI sequences of a score	every voice, played to a click	the Misa Criolla, The Rite of Spring, Petrushka, tango and folk arrangements
MIDI performances	a pianist playing freely, with no click	tangos played on the piano

Source	What it holds	For
A transcription in scale degrees (Hooktheory's TheoryTab)	the tune and the chords as numbers in a key	The Big Bang Theory theme
SymbTr	Turkish makam music, note by note	the Turkish songs
Page images of a printed score	what the eye reads, bar by bar	a few tangos and songs

The text formats are already close to the app's notation, so their scripts mostly translate. A lead sheet's triplets become the nearest sixteenths, the way the [lead sheet](#) itself rounds them, and where a lead sheet's chord clashes with its own melody the chord jazz musicians usually play there is used, with a word in the comment. Where a source gives no chords, the comment says the chords are a plain harmonization. Many Balkan dances are written in sixteenths (7/16, 11/16); here they are written in eighths, every note twice as long, which sounds the same at the tempo given.

Reading a score

An engraving, through MIDI

LilyPond is a language for printing music from text. Here are the first notes of the first violins in Handel's "For Unto Us a Child Is Born":

```
re'''4 sol''8 sol'' do''' do''' do''' do''' |
```

The note names are Italian (**re** is D, **sol** is G, **do** is C), each **'** raises the note an octave, and the number is the length (**4** a quarter, **8** an eighth). LilyPond writes a score as a MIDI file as well as a page, and the script reads that file bar by bar:

- **Dynamics.** LilyPond plays each dynamic mark at a set velocity (*p* 69, *mf* 86, *f* 95), so those velocities read back as the marks, **!p**, **!mf**, **!f**.
- **Staccato.** LilyPond plays a staccato note half as long as it is written. To tell a staccato quarter from an eighth, the score is compiled twice, once without its staccato dots: a note that is shorter in the first file is staccato, and gets a **.** in the demo.

- **Thirty-second notes.** Bach's written-out trills are faster than a step. The notation can write half a step, `D5:1/2`.

Some scores leave things out, and the comment says what was added: Messiah's edition prints its fortes in the first violins only, so a forte holds for all the strings. Editions also have mistakes. Die Forelle's engraving writes D natural in bars 43 and 44, against a D flat in the left hand. Bars 19 and 20 have the same music with D flat, and so does a second edition, OpenScore's: the demo plays D flat.

A pickup of its own length

Alfonsina y el mar starts one eighth note before its first full bar. Rather than a bar of 6/8 that is nearly all rest, as "Clementine" writes its pickup, the upbeat can be a bar of its own length, since a part can have its own time signature:

```
const ONE_EIGHT: TimeSignature = TimeSignature::new(1, 8);
```

```
Part::new("Pickup", "!mf F#4:2", "")
    .in_time(ONE_EIGHT)
```

`TimeSignature::new` is a `const fn`, so a time signature can be a constant too.

A choir is three channels, not one

Two voices of a choir often sing the same note. Say the altos start an E in sixteenths while the tenors hold an E for a whole beat: on one channel those are two notes with the same pitch on the same step, and a pattern never keeps two such notes (see [No two notes in the same place](#)): the first wins, and the tenors' held E would be lost. So the chorus has its altos, tenors and basses on three lines, three channels, as they are on three staves of the score. The Misa Criolla's choir is one line, "Choir", its voices joined by `&`: where two of them meet on one note, that note sounds once.

Figured bass: chords written as numbers

Baroque composers did not write out the keyboard player's chords. Under the bass line they wrote small numbers, the **figures**, each an interval above the bass note, and the player filled in the chords. Chrysander's edition of Messiah prints figures over some of the bass, and the Handel demo's chart follows them where they are given:

Figure	Means	In this chorus
none	a root-position triad	most bars
6	a triad in first inversion: the third in the bass	bar 1: B with 6 is G/B
6/5	a seventh chord in first inversion	bar 1: F# with 6/5 is D7/F#
4/2	a seventh chord in third inversion: the seventh in the bass	bar 28: G with 4/2 is A7/G ; bar 43: C with 4/2 is D7/C
6 with a slash	first inversion, the sixth raised a semitone	bar 26: E with a raised 6 is C#dim/E
+ on its own	the third raised	bar 22: A with C#, A
a natural sign	the third made natural	bar 58: D with F natural, Dm

A seventh wants to fall a step, so a 4/2 chord, with the seventh in the bass, is nearly always followed by a 6 on the note below: **A7/G** goes to **D/F#** in bar 29, **D7/C** to **G/B** in bar 44. How the chord sheet writes inversions is in the manual's [Slash chords and inversions](#).

Reading a sequence

A **sequence** is a MIDI file made in a sequencer, every voice on its own track, to a click. The Misa Criolla, the Rite of Spring and Petrushka come from sequences of their scores. Bar lines are in the file; what the script has to do is read the performance back as a score:

- each note goes on the nearest sixteenth, or the nearest third of one where the sequence plays a triplet, which then lands exactly on the tick (see [Notes between steps](#));
- the tempo map is read as the score's marks: a deep dip that comes back is a **fermata**, a run of small changes a **glide** (*rit.* or *accel.*), and anything else a new tempo;
- the velocities are read as dynamics, from *pp* to *ff*;
- a **tremolo**, the same chord struck faster than a sixteenth, is written as one held chord.

An orchestral score has more lines than a band has channels. The script keeps the tune in front and follows it from instrument to instrument with `played_by`, gives the lines that matter their own channels, and writes the rest on the Keys channel. Lines doubled at the unison or the octave are written once. Grace notes, trills and glissandos are simplified, and the comment says how.

Reading a performance

A pianist playing freely, with no click, leaves a MIDI file with no bar lines in it: the file's own grid means nothing. The tangos in `tango_played.rs` come from such files. Everything a score would say has to be heard.

Finding the beat

The beat is found the way a listener finds it, with the method of Dan Ellis's "Beat tracking by dynamic programming" (2007), the same one the app uses on a recording (see [Beats: onsets, autocorrelation and dynamic programming](#)): how much new sound starts at each moment, the steadiest period in it, then the chain of beats that lands on the most onsets while keeping close to that period. Each bar then gets the tempo it was played at, and a tempo marker is written wherever the tempo moves by half a beat a minute or more. Uno has 61 of them.

A steady beat off the grid

Some files are sequenced to a steady click but saved at another tempo, so the click's beats are a fraction of a tick long and drift across the file's own bars. Balderrama's notes fall nowhere near the file's grid, but its bombo strikes every 293 or 294 ticks from the first bar to the last: a steady beat. To find it exactly, number the strokes by the beat they fall on, `k = 0, 1, 2, ...`, and fit a straight line through them, tick against `k`. The **least squares** line is the one with the smallest sum of squared misses, and its slope, the length of the beat, has a formula:

$$\text{period} = \sum (k - \bar{k})(t - \bar{t}) / \sum (k - \bar{k})^2$$

where `$\bar{k}$` and `$\bar{t}$` are the averages. For Balderrama the eighth note is 293.966 ticks long, and the strokes miss the line by a few ticks, thousandths of a second: a machine, not a player.

The line gives the beat but not the bar: which of the six eighths is the first? Harmony changes on the downbeat, and of the choir's long held notes most start on one and the same eighth of the six. That eighth is the first of the bar, and the tune turns out to start two eighths before it, with a pickup, as a zamba often does.

The tune and the key

The **tune** is the top line: of the notes sounding at each moment on the melody's track, the highest (the **skyline**, as the [lead sheet](#) reads a tune too). Notes under it that double the tune, such as a second voice in thirds, go on a line of their own.

The **key** comes from how long each of the twelve pitch classes sounds, compared with key profiles, the way the app's own key finder does it (see [Comparing: the Krumhansl-Schmuckler method](#)). Where it chooses wrong the key is set by hand, and a part that changes key gets its own `in_key`. The test that at least four notes in five are in their part's key (`melodies_are_mostly_in_their_key`) catches a wrong choice of minor, too: Balderrama's tune sings D natural far more often than D sharp, so in E harmonic minor only 228 of its 288 notes are in the key and the test fails, while in E natural minor 257 are.

The chords

For every bar, half bar or beat, the script scores each common chord on all twelve roots: the notes inside the chord count for it, the notes outside against it, and a bass note on the chord's root counts extra. Chords of the key get a small bonus, so a passing note does not turn A minor into something exotic, and a bar is split only where the split scores clearly better. That is the same idea as the app's own [chord hearing](#), and it has the same weakness: notes alone name every colour as a chord of its own. A sixth the pianist adds turns Bm into Bm6, and a note of the tune on its way adds a flat ninth to F#7.

A **lead sheet** of the song fixes that. Its bars do not line up with a free performance (the pianist adds an introduction, repeats a strain, stretches a bar), so its chords are not laid over the bars one by one. Instead the sheet gives the reading its **vocabulary**: only the chords the sheet uses may be chosen. A sheet in another key is first **transposed**, every chord moved by the same interval, which keeps each chord's job in the key: Uno's sheet is in G and the pianist plays in C, so G becomes C, D7 becomes G7 and Em becomes Am.

Yira, yira's bars 38 to 40, read from the notes alone, are `Bm6 F#7b9 | Bm F#7 | Bm A7` . Limited to the sheet's chords they are

The left hand colours the B minor with its sixth, G#, and the tune touches G, a flat ninth over F#7. Under the colours are the chords the song is built on, i and V7 of B minor.

With no lead sheet, the chords are checked by ear and by what they do. Tiempos viejos ends on D, F, A and B. Those four notes are also B D F A, **Bm7b5**, but the bass plays D, and the chord comes after A7, the dominant of D minor: it is **Dm6**, D minor with an added sixth. A chord's name follows its bass and its job, not only its notes.

Spelling the notes

A MIDI file has numbers, not names. Note 70 is A sharp or B flat, and the file does not say which. The key names the seven notes of its scale; the other five follow the chord around them:

- **A note of the chord is spelled as the chord has it.** In F#7 (F#, A#, C#, E) note 70 is A sharp, the chord's third; in Gm (G, Bb, D) it is B flat.
- **A chord on a note outside the key is named by where it goes.** A dominant seventh falls a fifth, so the chord before C#m is G#7. A seventh chord that falls a semitone is a **tritone substitute**: the chord before G is Ab7, not G#7. A diminished seventh rises a semitone: G#dim7 goes to A.
- **A note outside the chord is spelled by where it goes.** A chromatic passing note between two notes a whole tone apart is a sharp on the way up and a flat on the way down: A, A#, B and G, Gb, F.
- **No E sharp, B sharp, C flat or F flat for a passing note:** on the page they look like mistakes for F, C, B and E. The key's own notes still can be one: in B minor the raised fourth that leads up to the dominant, F sharp, is E sharp.

The app's own spelling of notes outside the key, which needs no chord, is in [Notes from outside the key](#); the lead sheet's, which looks at the chord as these rules do, in [Which letter](#).

A transcription in scale degrees

The Big Bang Theory theme comes from a **transcription** on Hooktheory's TheoryTab site, where songs are written down the way a theory class reads them: every note a **scale degree** in the key (1 is the tonic, 5 the fifth), every chord a Roman numeral. The

Sheet Sage dataset (Chris Donahue and others, 2022) gathers about 26,000 of them in one file. A note there looks like this:

```
{"sd": "6", "octave": 0, "beat": 1, "duration": 1}
```

Scale degree 6, in the octave that starts on the tonic, on beat 1, a beat long. The theme is in G, and with the tonic as G4 the tune runs from D4 to G5, the range a lead sheet writes it in:

Written there	In G	Here
sd 6, octave 0, 1 beat	E, above the tonic	E5:4
sd 5, octave 0, ½ beat	D	D5:2
sd 1, octave 0, ¼ beat	G, the tonic	G4:1
sd 5, octave -1, ½ beat	D, below the tonic	D4:2

The transcription is also **aligned** to the recording: the verse’s beat 1 is at 7.6 seconds and its beat 33 at 19.66 seconds. 32 beats in 12.06 seconds is $32 \times 60 / 12.06 = 159.2$ beats a minute, so the verse carries `.with_tempo(&[tempo(0, 159)])` after an intro at 130.

What a demo must keep

Two tests over every song shape how a piece is cut into parts and how its tempo is written.

A MIDI export must come back note for note.

`every_demo_comes_back_note_for_note` exports each demo as a MIDI file, imports it, exports it again and compares the two files. The importer cuts a long song into patterns at bar lines no note crosses (see [Cutting the song into patterns](#)); where every bar line for many bars is crossed by a held note, it has to cut one, and the cut note is struck again. A demo strikes held notes again at its own part boundaries anyway, so a boundary at the right bar line makes a clean cut: the Misa Criolla’s Credo, whose bass ties over almost every bar line, has one at bar 30, where the soloists come in.

A demo keeps a sensible tempo. `every_demo_keeps_a_sensible_tempo` keeps every tempo marker between three tenths and two and a half times the song’s tempo. A sequence’s last chord is often held by slowing the click right down: the Credo’s sequence holds its last chord at 20 beats a minute, and the demo at 46.

One more test reads the songs in a way that needs care. The key finder's test measures how many bars of the demos it hears in their written key, and a piece in two keys at once would make that figure meaningless: whichever key it names, half the music disagrees. Such songs go on a list of their own:

```
const POLYTONAL: &[&str] = &["The Rite of Spring, The Augurs of Spring"];
```

The main test leaves them out, and a second one checks that the written key is among the keys heard, but not the only one:

```
assert!(  
    result.rightBars < result.bars,  
    "{name} is heard in one key: drop it from POLYTONAL"  
);
```

The second assert keeps the list honest: if the finder ever heard the Augurs in one key, the song would no longer belong there.

Checking a demo against its source

The last step is to play the demo with the [playback trace](#):

```
ferrous-lineup --trace "Balderrama" --out balderrama.tsv
```

and look for every note of the source in the trace: for a score at the same step and pitch, on the right channel and with its length; for a performance at the same pitch within a fifth of a second. What is missing must be what a demo leaves out on purpose:

- unisons on one channel, which sound once (two hands on the Keys channel, two guitars striking the same note, the voices of a choir on one line);
- tremolos and strums faster than a sixteenth, written as held chords;
- drum strokes that land on the same step of one lane;
- notes the comment names, such as an edition's mistake put right.

Exercises: songs as data

These exercises practise this part on the real songs: the constants in `library/song/`, the demo they become, and the music they were read from. Each one says what is **already there**, so you extend it rather than build it again. Most are things to open in the app and listen to, or to work out with a pencil; some are pieces of Rust with a clear end; the ones called “Think about it” are questions, with no code to write.

Try the Rust ones on a branch of your own. Run `cargo test` after each change, and `cargo fmt --all --check` and `cargo clippy --workspace --all-targets -- -D warnings` before you call one finished: CI runs all three.

Songs as constants

1. A new song in its genre file (lineup-core, medium)

Add a public-domain tune to the demos. `Song`, `Part`, `SONGS` and the genre files all exist; you are adding an entry, not a feature (see [A song is a constant](#)).

1. Read two songs in the genre file you chose, such as `folk.rs` or `blues.rs`, and copy the shape of one, every field from `name` to `about`.
2. Write the tune with `|` between bars and a chart under it, and give the part a name such as “Verse” or “Tune”.
3. Make it `pub(super) const`, and add it to `SONGS` in `song.rs`, at the end of its genre’s block.
4. Let the tests find the mistakes: a bar that does not add up, a chord that cannot be read, a style in another meter, a note that is not in the key.

```
cargo test -p lineup-core library
cargo run --release -- --demos
```

It is done when the tests pass and the song is under **File > Demos > Songs**, in its genre, opens on the chord sheet and plays.

Practises: struct literals, `const` data, reading test failures.

2. Extend a song that is already there (lineup-core, medium)

Shenandoah plays its song twice after a pickup, and **Danny Boy** a verse and a chorus. Pick one and add a part the demo does not play yet: an introduction for the band alone, a last verse, or a coda. Write it bar by bar, give it a chart, and add it to the song's `form`. Then open the demo and turn on the chord sheet: does the sheet show the chords you wrote, in the bars you meant?

Practises: the melody notation, `form` as a list of names.

3. A last chorus a semitone higher (lineup-core, easy)

Many pop songs lift their last chorus by a semitone. Add a copy of a song's chorus with `.in_key(...)` one semitone above the song's key, and put it last in the `form`. Which numerals of the chart have to change? (None: they are read in the part's key.) Play the end: the demo's last bar is on the home chord of the new key, as `a_song_that_ends_after_a_modulation_ends_in_the_new_key` checks.

Practises: builder methods on a `const` value.

4. A canon at the fifth (lineup-core, medium)

`with_voices` takes `(voice, bar)` pairs, so every voice of a round sings the tune at the same pitch. Many canons have the second voice a fifth higher or lower. Make each pair a triple, `(voice, bar, semitones)`, and move each voice's notes by its semitones where the demo builds the voices. Every round already written gets a `0`. Then try Frère Jacques with its second voice `7` semitones up: which notes clash, and why does a round need its copies at the same pitch to fit the same chord?

Practises: changing a type and following the compiler's errors.

5. Your song's first notes (lineup-core, easy)

`the_first_notes_are_right` checks that Ode to Joy starts F# F# G A and Twinkle, Twinkle starts C C G G. Add a line for a song you know well, with its first four notes as MIDI numbers and their names in the message. Get one wrong on purpose first, and read what the failure says.

Practises: `assert_eq!` with a message, MIDI note numbers.

6. Lean on the highest note (lineup-core, medium)

`perform` in `library/interpret.rs` gives a note of a tune up to six velocity steps for how high it is in its phrase. Singers often lean on the single *highest* note of a phrase more than on the notes near it. Give that one note two steps more, and write the test first: in Twinkle, Twinkle, which note is the highest of the first phrase? Keep the test `a_tune_gets_its_accents_but_stays_close_to_how_it_was_written` passing.

Practises: `Iterator::max_by_key`, writing the test before the code.

7. Search by meter and key (the app, easy)

Open **File > Demos** and type `7/8` in the search box, then `5/4`, then `2+2+3/8`. Then try `d minor`. Which matches come first, and which are fainter? Why does `fold` keep `/` and `+` when it drops every other mark (see [Search](#))?

8. Think about it: methods that take self

`in_key` and `with_tempo` are `const fn`s that take `self` by value and return a new `Part`. Why does that let them be chained inside a constant, `Part::new(...).in_key(...).with_tempo(...)`? Why would a method taking `&mut self` not work there?

Reading the music

9. Which black key? (pencil, easy)

In C major, name the black key between F and G, with the rules of [Spelling the notes](#): over a D7 chord; in the line G, ?, F; and over an Ab7 going to G. Then name the black key between A and B over Gm, and over F#7.

10. A new note, or a new mode? (pencil and the app, easy)

Write the scales of E natural minor, E harmonic minor and E Dorian. Which note differs in each? Then play bars 21 to 23 of **Balderrama**. The tune sings C#: over which chord? Is the music in E Dorian there, or is the C# a note on its way somewhere? And

why is the song written in E natural minor, not E harmonic minor (see [The tune and the key](#))?

11. Two ways to count six (the app, easy)

Clap six even claps, then group them 3+3 (ONE two three FOUR five six) and 2+2+2 (ONE two THREE four FIVE six). Play **Acuarela del río**, a polca in 6/8, and clap along with the bass, then with the drums. Which grouping does each play?

12. A pianist's tempo (the playback trace, easy)

Trace **Uno**:

```
ferrous-lineup --trace "Uno" --out uno.tsv
```

and read its `# tempo:` lines for the first eight bars. Where does the pianist slow down, and where in the phrase does it happen? How many times does the tempo change in the whole song?

13. Roots a fifth apart (the app, easy)

In **For Unto Us a Child Is Born**, play bars 41 and 42 with only the Keys channel unmuted. Write down the chords from the second beat of bar 41 and their roots. How far is each root from the next? The note the Demos menu shows for the song names the same chain.

14. Another 4/2 chord (the app, easy)

Bar 28 of the Handel chorus has `A7/G`, a seventh chord with its seventh in the bass, figured 4/2 (see [Figured bass](#)). Bar 45 has another. Which chord is it, which chord follows it, and is the next bass note a step below?

15. The trout's dominants (the app, easy)

Die Forelle is in D $\flat$ major, and its third verse is the one that changes. Find the four bars of E $\flat$ 7 on the chord sheet. What chord comes after them, and what is it the dominant of? And what is E $\flat$ 7 the dominant of?

16. Why C minor needs B natural (the app, easy)

In **Sheep May Safely Graze**, find the middle section's bars in C minor on the chord sheet. Which chord brings in B natural, and why does a minor key need it? (Compare the key's scale, the harmonic minor, with C natural minor.)

17. Who plays Boléro (the app, easy)

Play **Boléro** and write down, for each statement of the tune, the instrument that plays it: the pattern names on the Song track tell you. How many statements are there before the music leaves C major, and which key does it go to? Then read Boléro's parts in `classical.rs`: in "Horn, celesta and piccolos", `played_by` gives the piccolos 19 and 28 semitones. Which intervals above the horn's tune are those, and which notes of the harmonic series do they copy?

18. The Petrushka chord (pencil, easy)

Play C, E, G and F#, A#, C# together: two major triads a tritone apart, the Petrushka chord. Now play the octatonic scale C, D♭, E♭, E, F#, G, A, B♭, which alternates half and whole steps. Which of the scale's notes are not in the chord? Why can the chord not be in any one major or minor key?

19. A mode over a drone (the app, easy)

Open **Petrushka, The Shrovetide Fair in the Evening**, turn on 🔍 **Find the key** and play the Coachmen's Dance, written in E Mixolydian. Which key does the finder show? E Mixolydian has the notes of A major: why does a key finder that compares how long each note sounds hear the key with the same notes? Then print the key-finding report:

```
cargo test --release -p lineup-core --test key_finding report -- --ignored --nocapture
```

and find the last tableau of Petrushka in it. Which keys does the score give, and which does the finder hear in their place?

20. A chord the key does not have (the app, easy)

Tea for Two is in A♭ major. Find the key marker where its second part starts. Which chord of A♭ is C major? (Count up from A♭: C is the third note, so C major is III, a chord

the key of A $\flat$ does not contain.)

21. From minor to major (the app, easy)

Volver's verse is in G minor and its refrain in G major. Find the key marker where the refrain starts. Which two notes of G harmonic minor change in G major? Find them in the refrain's first bars.

22. Two tempos and four chords (pencil and the playback trace, easy)

Trace **The Big Bang Theory theme** and find the first note of bar 5 and of bar 6. How long is a bar of 4/4 at 159 beats a minute? ($4 \times 60 / 159 = 1.509$ seconds.) And at 130? Then:

- Name the four chords of the verse, IVadd9 V I vi7, in G and then in A.
- In bar 10 the tune alternates G and F $\sharp$ over G, then over Em7. Over G, F $\sharp$ is the chord's major seventh; what is it over E minor?

23. A tango against its lead sheet (research, medium)

Pick one of the tangos in `tango_played.rs`, such as **Patotero sentimental**, and find a lead sheet of it. Transpose its chords to the pianist's key, then compare one part bar by bar with the demo's chart. Which chords did the notes alone name by a colour (a sixth, a ninth) rather than the chord the song is built on?

24. Think about it: a script in the app

The scripts that read MIDI files and engravings are not part of the app. What would it take to make **File > Import MIDI file...** write a demo constant instead of a project? List what a person decides now that the importer would have to decide: the key, where the parts begin, which track is the tune, which chords a lead sheet allows. Which of those does the app already know how to guess (see [Hearing harmony](#))?

How the code is tested: unit tests, window tests, traces

Files: the `mod tests` at the end of most files in `crates/*/src`, `crates/lineup-core/tests/`, `crates/lineup-listen/tests/` and `crates/lineup-app/src/app/ux_tests.rs`

Ferrous Lineup has more than 1,300 tests: about 900 in `lineup-core`, 150 in `lineup-audio`, 260 in `lineup-app` and a few dozen in the other two crates. Every chapter of this book ends with the tests of what it describes. This one is about the kinds of test, why each exists, and how to run them.

Running them

```
cargo test           # every crate
cargo test -p lineup-core # one crate
cargo test frequency  # only tests whose name contains
                        "frequency"
cargo test -- --nocapture # show what the tests print
```

A test is an ordinary function marked `#[test]`. It passes if it returns, and fails if it panics, which is what `assert!`, `assert_eq!` and `unwrap` on an error do. `cargo test` builds the crates in a test mode, runs every test in parallel threads, and lists the ones that failed with their messages.

The tests normally run in a **debug** build, and the continuous integration runs them that way too. That matters: in a debug build an `+` or a `*` that overflows panics, and every `debug_assert!` is checked, while a release build wraps the number around and skips the `debug_assert!` s. A bug of that kind passes in release and fails in debug, so debug is where tests should run. The few crates that would make a debug build too slow to play music, such as the SoundFont synthesizer, are built optimized even then (see [Debug builds](#)). Long reports, below, are the exception: they run with `--release`.

Unit tests, next to the code

Most tests sit at the end of the file they test:


```
#[cfg(test)]
mod tests {
    use super::*;
```

`#[cfg(test)]` leaves the module out of every build except `cargo test`, so the tests cost the program nothing. `use super::*` brings in everything of the file, private items too, since a child module can see its parent's private items. So a unit test can check a helper the rest of the program never sees.

Sound can be tested without listening. A sine wave crosses zero twice in every cycle, so counting the times the signal changes sign measures its pitch:

```
fn sign_changes(samples: &[f32]) -> usize {
    samples
        .windows(2)
        .filter(|pair| (pair[0] < 0.0) != (pair[1] < 0.0))
        .count()
}
```

`windows(2)` gives every pair of neighbouring samples, `filter` keeps the pairs on either side of zero, and `count` counts them. One second of A4 must change sign about 880 times, twice 440:

```
// A4 is 440 Hz: about 880 sign changes in one second.
assert!((860..=900).contains(&changes), "got {changes}");
```

The range allows for the attack at the start of the note. Iterators are lazy: nothing runs until `count` asks for values, and the compiler turns the chain into a loop as fast as one written by hand.

Tests over data

Much of Ferrous Lineup is data: 343 songs, 215 harmony styles, 139 rhythms, the twelve-tone and micromode styles. Tests read all of it, so a mistake in a table fails before anyone hears it: a bar that does not add up, a chord that cannot be read, a name used twice, a style whose rhythm does not exist (see [Testing it](#) for the library and [Testing it](#) for the songs). Their messages name the entry, so a failure says which song to open.

Tests in tests/, from the outside

A file in a crate's `tests/` folder is an **integration test**: a program of its own that uses the crate as any other crate would, so it sees only what the crate makes public.

File	What it checks
<code>lineup-core/tests/chord_reading.rs</code>	how well the chord sheet names what the hardest demos play: the Augurs chord, the Petrushka chord, a stack of fourths
<code>lineup-core/tests/key_finding.rs</code>	how many bars of the demos the key finder hears in their written key, and that it finds clear modulations on their bar
<code>lineup-core/tests/damaged_files.rs</code>	that a damaged project, MIDI file or chord sheet ends in an error or a project, never in a crash
<code>lineup-listen/tests/key_changes.rs</code>	how well key changes are found in demos rendered to audio

Damaged files

`damaged_files.rs` reads hundreds of damaged copies of good projects, MIDI files and chord sheets, as a **fuzzer** does, and checks that each ends in a project or an error, never in a crash. How it damages them, and the overflow it found that only a debug build sees, are in [Damaged files never crash the app](#).

Reports

Some tests measure rather than check. `report` in `key_finding.rs` and `chord_reading.rs` prints a table, song by song, of how well each demo is heard. They are marked `#[ignore]`, so `cargo test` skips them, and run on their own:

```
cargo test --release -p lineup-core --test key_finding report -- --ignored --nocapture
```

The ordinary tests next to them keep the figure from falling:

`most_bars_of_the_demos_are_heard_in_their_key` fails if fewer than 87% of the bars are heard in their key. The report says why a change moved that figure; the test makes sure it never moves the wrong way unnoticed (see [How well it hears](#)).

Window tests

egui draws the whole window again on every frame, from the app's state (see [Immediate mode](#)). That makes the window itself easy to test: a test can run frames without a screen, feed them clicks and keys, and read what was drawn. `ux_tests.rs` keeps a `Screen` for that:

```
fn new() -> Self {
    let ctx = egui::Context::default();
    theme::apply(&ctx);
    ctx.all_styles_mut(|style| style.animation_time = 0.0);
    let mut app = LineupApp::with_audio(&ctx, None,
    Audio::Unavailable("test".into()));
```

The app starts as it does without a sound card (`Audio::Unavailable`), with animations off so one frame shows the end of every animation, and it never reads or writes a real recovery copy or downloads a SoundFont. `draw` runs one frame with the events given; egui returns the shapes it would paint, and `text_rect` finds a label among them:

```
fn text_rect(output: &FullOutput, label: &str) -> Option<Rect> {
    output
        .shapes
        .iter()
        .rev()
        .find_map(|shape| match &shape.shape {
            Shape::Text(text) if text.galley.text() == label => {
```

`click` draws a frame, finds the label, and sends a pointer press and release at its centre. So a test reads like a person using the app:

```
fn f1_and_the_help_menu_show_the_keyboard_shortcuts() {
    let mut screen = Screen::new();
    let output = screen.draw(vec![]);
    assert!(text_rect(&output, "Keyboard shortcuts").is_none());

    screen.key(Key::F1);
    let output = screen.draw(vec![]);
    assert!(screen.app.shortcuts_open);
```

126 tests in that file open demos, apply styles, click the mixer and the rack, resize windows and restart the app. Smaller ones sit next to a single widget, such as the time signature picker or the playlist, with a `frame` helper that draws only that widget in a window of its own.

Traces: what was played

A test of the sequencer can count notes, but a song's sound is what the player sends to the instruments, step by step, at the tempo of the moment. The [playback trace](#) writes exactly that, one line per note, playing headless as **File > Export WAV...** does. It is the tool for a question a unit test cannot ask easily, such as "does this demo still play the same as before?": trace it before and after a change and compare the two tables. MIDI is tested the same way, without a synthesizer: a song is exported to bytes, read back and compared (see [Testing without a synth](#)).

Formatting and lints

Two more tools keep the code tidy, and the continuous integration runs both:

- `cargo fmt` rewrites the code in the standard Rust style, so nobody argues about where braces go. `cargo fmt --all --check` only says whether anything would change.
- `cargo clippy` is Rust's **linter**. It knows hundreds of patterns that compile but could be simpler, faster or more correct, and explains each one. `-D warnings` makes every warning an error.

```
cargo fmt --all --check
cargo clippy --workspace --all-targets -- -D warnings
```

`--all-targets` lints the tests and build scripts too. The continuous integration installs the newest stable Rust on every run, and a new Rust often brings new lints, so run `rustup update` before you lint: a lint your older Clippy does not know will fail the run anyway.

Three systems and continuous integration

Files: `.github/workflows/ci.yml`, `crates/lineup-app/src/main.rs`, `crates/lineup-app/build.rs` and `Cargo.toml`

Ferrous Lineup runs on Linux, macOS and Windows from one code base. Almost none of its code knows which system it is on, and three workflows on GitHub check every change on all three. This chapter is about how a Rust program stays portable, and how the checks run.

Who talks to the operating system

Ferrous Lineup never calls the operating system for sound, windows or dialogs itself. Four crates do:

Job	Crate	Linux	macOS	Windows
Sound	<code>cpal</code>	ALSA (also through PipeWire or PulseAudio)	Core Audio	WASAPI
Window, keys and mouse	<code>winit</code> , inside <code>eframe</code>	X11 or Wayland	AppKit	Win32
Drawing	<code>wgpu</code> , inside <code>eframe</code>	Vulkan or OpenGL	Metal	DirectX 12 or Vulkan
File dialogs	<code>rfd</code>	the desktop portal	the Finder's open and save panels	the Explorer dialog

Each of these crates holds the code for every system and keeps only the right one while compiling. The app calls `cpal::default_host()`, draws with egui and opens an `rfd::FileDialog`; none of that names a system. Each library was chosen for that: anything that worked on only one system would have to be replaced to reach the others.

The workspace's own code names a system in four places:

- `main.rs` hides the console window on Windows (below);
- `build.rs` puts the icon and the version into the Windows `.exe`;
- the lead sheet chooses `cmd`, `open` or `xdg-open` to show a PDF in the system's viewer;
- the sounds window shows a folder inside the home folder as `~/...` only on Linux and macOS, where people write paths that way.

How a crate keeps one system's code: `cfg`

Inside `cpal`, the default host is chosen like this (shortened):

```
#[cfg(any(target_os = "linux", target_os = "freebsd"))]
mod platform_impl {
    pub fn default_host() -> Host {
        AlsaHost::new().expect("the default host should always be
available").into()
    }
}

#[cfg(target_vendor = "apple")]
mod platform_impl {
    pub fn default_host() -> Host {
        CoreAudioHost::new().expect("the default host should always be
available").into()
    }
}
```

Two modules with the same name would normally be an error. But `#[cfg(...)]` is a **conditional compilation** attribute: when its condition is false, the item under it is removed before the compiler even looks at it. On Linux only the first module exists; on a Mac only the second. Conditions can ask about:

- the system you build for: `target_os = "windows"`, `target_os = "macos"`, `target_family = "unix"` (or just `unix`), `target_vendor = "apple"`;
- the kind of build: `debug_assertions` is on in debug builds, `test` while running `cargo test` (every `#[cfg(test)] mod tests` uses it);
- a crate's **features**: `feature = "wayland"`;
- and combinations, with `any(..)`, `all(..)` and `not(..)`.

Cargo can do the same for dependencies. `cpal`'s own `Cargo.toml` asks for the ALSA library only on systems that have it:

```
[target.'cfg(any(target_os = "linux", target_os =  
"freebsd"))'.dependencies]  
alsa = "0.11"
```

There is also a `cfg!(...)` macro, which gives `true` or `false` as an ordinary value. The lead sheet uses it:

```
let mut command = if cfg!(target_os = "windows") {
```

The difference matters. With `cfg!`, every branch is still compiled, so every branch must be valid on every system; with `#[cfg]`, code for other systems is never compiled at all, which is why it can call functions that only exist there. That is also its danger: a typo in a `#[cfg(windows)]` function goes unnoticed on Linux, because the compiler never reads it. Code you cannot compile is code you cannot trust, which is why it is best kept inside a few well-tested crates, and why the continuous integration below builds on every system.

Features: choosing the parts of a crate

The `rfd` line in the workspace's `Cargo.toml` turns part of the crate off:

```
rfd = { version = "0.17", default-features = false, features = ["xdg-  
portal"] }
```

A **feature** is an optional part of a crate, named in its `Cargo.toml`. Some are on unless you say otherwise: the `default` features. `rfd` normally turns on two: `xdg-portal`, which asks the Linux desktop to show its own file dialog (over D-Bus, the desktop's message system, so there is no GTK to install), and `wayland`, which tells the portal which window the dialog belongs to. Ferrous Lineup doesn't tell it, so `default-features = false` drops `wayland`, and `features = ["xdg-portal"]` keeps the one that is needed. Fewer features means fewer crates to compile. On macOS and Windows the Linux feature changes nothing: the `#[cfg]`s inside `rfd` leave its code out.

Features can only *add*. If two crates in a program use `rfd`, one with `wayland` and one without, Cargo builds it once, with `wayland`. So a feature must never take something away: a crate with it must still do everything it did without it.

No console window on Windows: `cfg_attr`

On Windows, every program is either a *console* program, which opens a terminal window when started, or a *windows* program, which does not. Rust makes console programs unless told otherwise, so double-clicking `ferrous-lineup.exe` would open an empty black window next to the app. One line at the top of `main.rs` says otherwise:

```
#![cfg_attr(not(debug_assertions), windows_subsystem = "windows")]
```

- `windows_subsystem = "windows"` asks for the second kind. On other systems the attribute does nothing.
- `#![...]`, with `!`, applies to the thing it is inside, here the whole crate. `#[...]` without `!` applies to the next item.
- `cfg_attr(condition, attribute)` applies the attribute only when the condition is true. `not(debug_assertions)` is true in release builds.

So `cargo run`, a debug build, keeps the console, where `println!` and panic messages show while you work, and `cargo build --release` makes the quiet program you give to other people. Those messages then go nowhere on Windows, which is one more reason the app shows audio errors and the “Could not open” and “Could not save” messages inside its own window.

A build script for the `.exe`

`crates/lineup-app/build.rs` is a **build script**: Cargo compiles and runs it before compiling the crate. On Windows it puts the icon, the version and the copyright into the `.exe`, so Explorer and the taskbar show the app’s icon and the file’s Properties page its name:

```
#[cfg(windows)]
{
    let version = env!("CARGO_PKG_VERSION");
```

Here `#[cfg(windows)]` describes the computer doing the build, because a build script runs there: the resource compiler it calls is Microsoft’s, which only Windows has. The release workflow builds the Windows program on Windows, so the icon is always there. macOS and Linux take their icons from the bundles and desktop files in `packaging/` (see [Release packages](#)).

Small differences the code already handles

- **Shortcuts:** `Modifiers::COMMAND` is Ctrl on Linux and Windows and ⌘ on macOS.
- **Paths:** `PathBuf` and `Path` handle `/` and `\`, drive letters and file names that are not valid text. `std::env::args_os` passes the file to open exactly as the system gave it.
- **Replacing a file:** `std::fs::rename` replaces an existing file on every system, which [safe saving](#) depends on.
- **Sound:** cpal hands each system's audio callback the same `&mut [f32]`, so the audio thread never knows where it runs.

Three workflows

A **workflow** is a file in `.github/workflows/` that tells GitHub what to run, on which machines, and when. Ferrous Lineup has three:

Workflow	Runs on	Does
CI (<code>ci.yml</code>)	every pull request and every push to main	formats, lints, tests and builds on Linux, macOS and Windows
Release (<code>release.yml</code>)	every pull request, push to main and <code>v</code> tag, and by hand	builds the downloads for every system, and publishes them for a tag, pushed or written in a run by hand (see Release packages)
Book (<code>book.yml</code>)	changes to the books and manuals	builds the websites and PDFs, and publishes them from main (see How the books are built)

A pull request shows a check for each job, green when it passed and red when it failed, with a link to its log.

One job, three machines: the matrix

The CI workflow has one job, `check`, run three times:

```

check:
  name: Build and test (${ matrix.name })
  strategy:
    fail-fast: false
  matrix:
    include:
      - name: Linux
        os: ubuntu-latest
        artifact: ferrous-lineup-linux
        binary: target/release/ferrous-lineup
      - name: macOS
        os: macos-latest
        artifact: ferrous-lineup-macos
        binary: target/release/ferrous-lineup
      - name: Windows
        os: windows-latest
        artifact: ferrous-lineup-windows
        binary: target/release/ferrous-lineup.exe
  runs-on: ${ matrix.os }

```

- A **matrix** turns one job into several. Each entry under `include` is one copy of the job, with its own values for `name`, `os`, `artifact` and `binary`.
- `${ matrix.os }` is replaced by the entry's value, so `runs-on` picks a Linux, Mac or Windows machine, and the job's name says which ("Build and test (macOS)").
- `fail-fast: false` lets the other copies finish when one fails. Without it, a Windows failure would cancel the Mac job, and a second problem would show only after the first was fixed.

Each copy then runs the same steps:

1. **Install system libraries**, on Linux only: the development files for ALSA, xkbcommon and Wayland. macOS and Windows come with theirs.
2. **Rust**, the newest stable, with `rustfmt` and `clippy`.
3. **A cache** of Cargo's downloads and compiled dependencies (`Swatinem/rust-cache`), so a run compiles only what changed.
4. **Check formatting**, on Linux only: formatting is the same everywhere.
5. **Lint**: `cargo clippy --workspace --all-targets -- -D warnings`.
6. **Test**: `cargo test --workspace`, in a debug build (see [Running them](#)).
7. **Build release**, and upload the program as the run's artifact.

The steps that only make sense on one system say so:

```

- name: Install system libraries
  if: runner.os == 'Linux'

```

Clippy and the tests run on all three: clippy because each system compiles slightly different code, and the tests because a test that passes on one system can fail on another (a path written with `/`, a file that cannot be renamed while it is open).

On the page of a run, under **Artifacts**, are the three programs. GitHub zips each, and the zip forgets that the program was executable, so on Linux and macOS it needs `chmod +x ferrous-lineup` after unzipping. The proper downloads, with installers, come from the Release workflow.

Checking Windows code from Linux

Waiting for CI to find a Windows mistake is slow. Rust can check code for another system without leaving Linux:

```
rustup target add x86_64-pc-windows-gnu
cargo clippy --workspace --all-targets --target x86_64-pc-windows-gnu -- -D warnings
```

A **target** is the system you build *for*, as opposed to the one you build *on*. `rustup target add` downloads Rust's standard library compiled for Windows. `cargo clippy` and `cargo check` only run the compiler's checks and never link a program, so they need neither Windows' linker nor its system libraries. That catches most mistakes, a Windows-only `#[cfg]` that does not compile or a crate that does not support Windows at all, though not what only happens when the program runs. The same works for Macs with `aarch64-apple-darwin` (Apple's processors) or `x86_64-apple-darwin` (Intel Macs), as long as no dependency needs Apple's own tools to build.

What the checks cannot prove

The CI machines have no sound card and no screen. The jobs prove that the program builds and that the music logic is right on each system, not that you hear anything: the window tests draw into memory (see [Window tests](#)), and the audio tests render into a buffer. A run on a real Mac or Windows computer is still the real test of sound and drawing.

Release packages

Files: `.github/workflows/release.yml` and the folders of `packaging/`

Most people who use Ferrous Lineup will never install Rust. The **Release** workflow builds the program for four kinds of computer and packs seven downloads:

Package	System
<code>ferrous-lineup-...-linux-x86_64.AppImage</code>	Linux on a PC (Intel or AMD): one file, nothing to unpack
<code>ferrous-lineup-...-linux-x86_64.tar.gz</code>	Linux on a PC, as a plain folder
<code>ferrous-lineup-..._amd64.deb</code>	Debian, Ubuntu and their derivatives
<code>ferrous-lineup-...-windows-x86_64-setup.exe</code>	Windows on a PC, with an uninstaller and the <code>.lineup</code> file association
<code>ferrous-lineup-...-windows-x86_64.zip</code>	Windows on a PC, portable: unzip and run
<code>ferrous-lineup-...-macos-arm64.dmg</code>	Macs with Apple Silicon (M1 and later)
<code>ferrous-lineup-...-macos-x86_64.dmg</code>	Macs with an Intel processor

The `...` is the version for a release (`v0.4.1`), or the first seven characters of the commit for any other build (`1a2b3c4`), so a download always says what it is. How to open each one on its system is in the manual's [Installing it](#); this chapter is about how they are made.

Where the downloads are

- **Every pipeline** runs the workflow: each pull request, each push to main, and each run by hand (**Actions > Release > Run workflow** builds any branch). The packages are the run's **artifacts**, at the bottom of its page. Artifacts need a GitHub login to download, and GitHub deletes them after 90 days. So a pull request can be tried on a Mac or on Windows before it is merged.
- **A tag** that starts with `v` also publishes the packages on the repository's **Releases** page, with a `SHA256SUMS.txt` beside them. A release stays there for

good, and anyone can download it. The tag can be pushed, or the workflow can make it itself in a run by hand (see below).

Cutting a release

A release is a tag: a name stuck on one commit. Making version 0.5.0 takes a pull request and a tag.

The version pull request changes:

- the version in the root `Cargo.toml`, under `[workspace.package]`, which every crate takes with `version.workspace = true`;
- `Cargo.lock`, which lists the version of every crate, ours too (one `cargo build` updates it);
- the release notes in the AppStream file, `packaging/linux/io.github.naelolaiz.FerrousLineup.metainfo.xml`, in English and Spanish, which software centres show;
- the default version in the Windows installer's script, `packaging/windows/ferrous-lineup.iss`;
- the version in the AUR recipes and the tag in the Flathub manifest;
- if the release should open with a few words of its own, a file `packaging/release-notes/v0.5.0.md` holding them, in Markdown.

The tag, once that is merged, comes one of two ways. From a clone, push it:

```
git fetch origin
git tag -a v0.5.0 -m "Ferrous Lineup 0.5.0" origin/main
git push origin v0.5.0
```

Or, on GitHub, open **Actions > Release > Run workflow**, keep the branch on `main`, write `v0.5.0` in the **tag** field and run it. This needs only the right to run workflows, not to push to the repository, and the workflow makes the tag itself, on the commit it built.

Either way, when all the build jobs have passed, the release appears, called “Ferrous Lineup v0.5.0”. Its notes are the release-notes file, if there is one, followed by the list of pull requests merged since the last release that GitHub writes; they can be edited on GitHub afterwards. A last small pull request pins what only exists once the tag does: the checksum of GitHub’s archive of the tag in the AUR recipe, and the tag’s commit in the Flathub manifest.

Three details:

- A tag with a dash, such as `v0.5.0-beta.1`, makes a **pre-release**, never shown as the latest. That is how [semantic versioning](#) writes a version that is not final yet, and Cargo reads versions the same way.
- If a pushed tag and `Cargo.toml` disagree, the release is still made, with a warning on the run's page: the tag exists already, and stopping would only leave it without downloads. The tag decides the file names, but the program's own **Help > About** says what `Cargo.toml` said.
- A run by hand checks before it makes anything: if the tag asked for is not `v` plus the version in `Cargo.toml`, or the run is on a branch other than main, the release job stops with an error, and no tag is made.

Targets: which computer a program is for

A compiled program is machine code for one kind of processor, in the file format of one operating system. Rust names each combination with a **target triple**, architecture-vendor-system(-environment):

Target	Processor	System
<code>x86_64-unknown-linux-gnu</code>	64-bit Intel or AMD	Linux, with the GNU C library
<code>x86_64-pc-windows-msvc</code>	64-bit Intel or AMD	Windows, with Microsoft's tools
<code>aarch64-apple-darwin</code>	64-bit ARM (Apple Silicon)	macOS ("Darwin" is its core)
<code>x86_64-apple-darwin</code>	64-bit Intel	macOS

`rustc --version --verbose` shows your own computer's target on its `host:` line. `cargo build --target <triple>` builds for another one, once `rustup target add <triple>` has fetched the standard library for it, and puts the program in `target/<triple>/release/`, so builds for different targets never mix.

In the workflow each target is a line of a matrix (see [One job, three machines](#)):

```
- name: Linux x86_64
  os: ubuntu-22.04
  target: x86_64-unknown-linux-gnu
  platform: linux-x86_64
  appimage: true
```

and the build is

```
cargo build --release --locked --target ${matrix.target }
```

`--locked` builds exactly the library versions in `Cargo.lock`, and stops with an error if the file would have to change. A release should be built from what was tested, not from whatever versions are newest on the day of the tag. That is also why the version pull request must commit `Cargo.lock`: with the old file, `--locked` refuses to build.

The Intel Mac build, made on an Apple Silicon Mac

GitHub’s macOS machines are Apple Silicon. Building for another processor than the one you build on is **cross-compiling**, and on a Mac it is easy: Apple’s compilers and system libraries cover both processors. So the Intel job runs on the same kind of machine as the Apple Silicon one; only its target differs. The bits of C in the build, such as the cryptography in `ring` (for downloading the SoundFont over HTTPS), are compiled by the `cc` crate, which passes the target on to Apple’s C compiler. The workflow can’t run the Intel program, which is fine: the tests run in CI, and the code is the same.

Why Linux builds on an older Ubuntu

A Linux program calls the C library, glibc, for files, memory and threads. Each glibc function carries a version, and a program records the newest one it used: built on Ubuntu 24.04, Ferrous Lineup would ask for `GLIBC_2.39`. An older glibc refuses to start it (“version ‘GLIBC_2.39’ not found”), while a newer one runs it happily. So the Linux build runs on the oldest Ubuntu GitHub offers, 22.04, with glibc 2.35, and the program runs on Ubuntu 22.04, Debian 12 and anything newer.

`ldd` lists the libraries a program loads when it starts:

```
$ ldd target/release/ferrous-lineup
libasound.so.2 => /lib/x86_64-linux-gnu/libasound.so.2
libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1
libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
```

Only ALSA is there. The window libraries (Wayland or X11, xkbcommon for the keyboard, OpenGL) are not missing: the crates open them while the program runs, with `dlopen`, and use whichever the desktop has. That is why one Linux build works on both Wayland and X11 desktops.

The C runtime inside the Windows program

Rust programs on Windows use Microsoft's C runtime for a few basics. By default it is a separate DLL that comes with the "Visual C++ Redistributable", which most Windows computers have, but not all. The Windows line of the matrix adds a compiler flag:

```
rustflags: -C target-feature=+crt-static
```

`crt-static` copies the runtime into the `.exe` (which grows a little), so the program starts on a Windows that never installed anything. The `RUSTFLAGS` environment variable passes flags to every `rustc` Cargo runs; on the other lines it is empty.

Packing

For Linux and Windows, a **Package** step copies the program, the README and the licence into a folder named after the label, and packs it:

```
if [[ -n "$RELEASE_TAG" ]]; then
  label="$RELEASE_TAG"
else
  label="${COMMIT::7}"
fi
name="ferrous-lineup-$label-${{ matrix.platform }}"
```

`RELEASE_TAG` is set once, for every job, at the top of the file:

```
env:
  RELEASE_TAG: ${{{ github.ref_type == 'tag' && github.ref_name ||
  inputs.tag }}
```

GitHub's expressions have no "if ... else", so `a && b || c` stands in for it: when the run is for a pushed tag, the value is the tag's name; otherwise it is the **tag** field of a run by hand, which is empty in every other run (`inputs` holds nothing when nobody filled a form). `-n` in Bash is true for a string that isn't empty.

`${COMMIT::7}` is Bash for "the first seven characters". `COMMIT` is `${{ github.event.pull_request.head.sha || github.sha }}`: for a pull request GitHub builds a merge commit of its own, which nobody ever sees, so the label takes the branch's latest commit instead; `||` picks the first value that isn't empty. Windows gets a `.zip`, which it opens by itself. Linux gets a `.tar.gz`, because a tar file keeps the program's "executable" permission.

For macOS, **Package for macOS** builds a real `Ferrous Lineup.app`: the program in `Contents/MacOS`, `Info.plist` and the icon from `packaging/macos/`, the version from `Cargo.toml` written into the plist. It signs the app **ad-hoc** (`codesign --force --sign -`), puts it in a folder with a link to `/Applications`, and `hdiutil` makes the `.dmg`, so people drag the app onto Applications.

The plist also declares the `.lineup` **document type**: a `CFBundleDocumentTypes` entry naming the type the app opens, and a `UTExportedTypeDeclarations` entry that defines that type, `io.github.naelolaiz.FerrousLineup.project`, as the files ending in `.lineup`. With both, the Finder opens a project with the app. On Linux and Windows the file manager starts the program with the file's path on the command line, which `main` reads. macOS starts it with no arguments, and sends the files in an Apple event, "open documents" (`aevt / odoc`), which also reaches an app already running. AppKit's own handler for it asks the application's delegate to open the files; the delegate belongs to `winit`, the window library, and opens nothing. So `open_files.rs` sets its own handler, at the moment Apple's rules give for it, when the application is about to finish launching (after AppKit sets its handler, before the event of the launch arrives). The handler only collects the paths:

```
#[unsafe(method(openDocuments:withReplyEvent:))]
fn open_documents(
    &self,
    event: &NSAppleEventDescriptor,
    _reply: &NSAppleEventDescriptor,
) {
    // SAFETY: `paramDescriptorForKeyword:` takes a four-letter
    // code and gives a descriptor or nil.
    let files: Option<Retained<NSAppleEventDescriptor>> =
        unsafe { msg_send![event, paramDescriptorForKeyword:
DIRECT_OBJECT] };
    let Some(files) = files else {
        return;
    };
    // A list of files, counted from 1, or a single one.
    let count = files.numberOfItems();
    let items: Vec<Retained<NSAppleEventDescriptor>> = if count == 0 {
        vec![files]
    } else {
        (1..=count)
            .filter_map(|index| files.descriptorAtIndex(index))
            .collect()
    };
    super::arrived(
        items
            .iter()
            .filter_map(|item| item.fileURLValue())
            .filter_map(|url| url.to_file_path()),
    );
}
```

`objc2` calls Objective-C from Rust: `define_class!` makes a class whose methods AppKit can call, and `msg_send!` sends a message by name, which the compiler cannot check, hence `unsafe`. `arrived` keeps the paths in a `Mutex<Vec<PathBuf>>` and asks the window to repaint; `handle_dropped_files` takes them on the next frame and opens the file as if it had been dropped on the window. All of it is under `#[cfg(target_os = "macos")]`, so on Linux and Windows `listen` does nothing.

Every package is uploaded with `archive: false`:

```
- uses: actions/upload-artifact@v7
  with:
    name: ${ env.NAME }
    path: ${ env.PACKAGE }
    archive: false
```

By default an artifact is zipped on upload, so a `.tar.gz` would download as a zip holding a `.tar.gz`. With `archive: false` the file is stored as it is, and the artifact is named after the file.

The AppImage

An **AppImage** is a program and what it needs, packed into one file that starts like a program. The file is two things glued together: a small starter program, the **runtime**, followed by a compressed disk image of a folder, the **AppDir**. Started, the runtime mounts the image as a folder (with FUSE, which lets a normal program provide a file system) and starts the program inside. Nothing is installed, so deleting the file uninstalls it. The AppDir is laid out like a system's `/usr`:

```
AppDir/
├─ AppRun -> usr/bin/ferrous-lineup          what the runtime starts
├─ io.github.naelolaiz.FerrousLineup.desktop -> usr/share/... the menu
entry
├─ io.github.naelolaiz.FerrousLineup.svg -> usr/share/... the icon
├─ usr/
│   ├─ bin/ferrous-lineup
│   ├─ lib/libxkbcommon.so.0, libxkbcommon-x11.so.0, libxcb-xkb.so.1
│   └─ share/applications/, icons/, mime/, metainfo/
```

The desktop entry

([packaging/linux/io.github.naelolaiz.FerrousLineup.desktop](#)) is the text file every Linux menu reads: the app's name, its icon, its category and how to start it (`Exec=ferrous-lineup %f`, where `%f` is a file to open). The MIME type file teaches the desktop what a `.lineup` file is, and the AppStream **metainfo** file describes the app and its releases for software centres. The workflow checks all three with `desktop-file-validate` and `xmllint`, because a typo in them fails silently on a desktop.

`appimagetool` turns the AppDir into the AppImage. It and the runtime are downloaded pinned to one version and checked against the SHA-256 written in the workflow:

```
echo "$RUNTIME_SHA256 runtime" | sha256sum --check
```

If someone replaced the file on GitHub, the checksum would differ and the build would stop, rather than ship whatever the file had become.

Bringing a library along

On a system without `libxkbcommon-x11`, the program stops before it opens a window. `ldd` doesn't list that library, because `winit` opens it with `dlopen` when the window is made. A library opened this way is searched for like any other: in the folders of `LD_LIBRARY_PATH`, then the **RUNPATH** written in the program, then the system's folders. So the workflow copies the three keyboard libraries into `AppDir/usr/lib` and writes a RUNPATH with `patchelf`:

```
patchelf --set-rpath '$ORIGIN/../lib' AppDir/usr/bin/ferrous-lineup
patchelf --set-rpath '$ORIGIN' AppDir/usr/lib/libxkbcommon-x11.so.0
```

`$ORIGIN` means “the folder this file is in”, worked out when the program starts, so the path holds wherever the AppImage is mounted. The single quotes keep the shell from replacing `$ORIGIN` with an empty variable. What the AppImage does **not** carry: glibc, ALSA and the graphics drivers. Those must match the computer, and the AppImage community keeps a list of libraries an AppImage should never bring.

The Debian package

The `.deb` is built by `cargo-deb`, pinned to one version, from the same program and the same desktop entry, icon, MIME type and AppStream file:

```
cargo deb -p lineup-app --no-build --target x86_64-unknown-linux-gnu
```

`--no-build` takes the program the matrix just built. The `[package.metadata.deb]` table in `crates/lineup-app/Cargo.toml` lists the files and what the package depends on (ALSA, Wayland, xkbcommon, the desktop portal and the tools that register menu entries, icons and MIME types), so `apt` installs those too.

The Windows installer

The Setup program is made by **Inno Setup**, which GitHub’s Windows machines already have, from the script `packaging/windows/ferrous-lineup.iss`. The workflow passes it the version and the label:

```
"$iscc" /DAppVersion="$version" /DBuildLabel="$LABEL"
packaging/windows/ferrous-lineup.iss
```

The script installs the program for the user who runs it, with no administrator rights needed, and adds a Start-menu entry, an optional desktop shortcut, an uninstaller and the `.lineup` file association.

Publishing: a job that waits for the others

The second job, `release`, runs only for a pushed tag or a run by hand with a tag, and only once every build job has passed:

```
release:
  name: Publish the release
  if: github.ref_type == 'tag' || (github.event_name ==
'workflow_dispatch' && inputs.tag != '')
  needs: build
```

`workflow_dispatch` is the name GitHub gives a run by hand. The form it shows comes from the trigger at the top of the file:

```
workflow_dispatch:
  inputs:
    tag:
      description: "Publish a release with this tag (v0.4.1), made on the
commit built. Empty: only build."
      required: false
      default: ""
```

`needs: build` waits for every line of the matrix. If one build fails, there is no release at all, rather than a release missing Windows.

The job's `permissions` are about the **token** every run gets, `GITHUB_TOKEN`, a password that expires when the job ends. At the top of the file, `permissions: contents: read` lets every job read the code and nothing else, so a build step cannot change the repository. Only this job asks for `contents: write`, which a release needs.

It downloads the packages, writes their checksums with `sha256sum`, and hands everything to `gh`, GitHub's command-line tool:

```
if gh release view "$TAG" > /dev/null 2>&1; then
  gh release upload "$TAG" release/* --clobber
else
```

- `gh release view` succeeds when the release exists already, made on the website or by an earlier run. Then the files are uploaded to it, and `--clobber` replaces files of the same name.
- Otherwise `gh release create` makes it, with `--generate-notes` for the notes. For a pushed tag it adds `--verify-tag`, which stops if the tag isn't on GitHub instead of creating a new tag on the newest commit. For a run by hand it adds `--target "$GITHUB_SHA"`, and GitHub makes the tag on exactly the commit the packages were built from, even if main has moved on since.
- `--notes-file packaging/release-notes/<tag>.md` is added when that file exists; `gh` puts its text before the generated notes.
- `--prerelease` is added when the tag has a dash.

A tag made with the run's own `GITHUB_TOKEN` starts no other workflow: what a run does with its own token starts no new runs (only an explicit “run by hand” through GitHub's API is let through), so that a workflow can't set itself off forever. That is why the run that makes the tag is the one that publishes it; nothing else will.

A **checksum** is a 64-character fingerprint of a file's bytes: change one byte and the fingerprint is completely different. `SHA256SUMS.txt` lets anyone check that a download arrived whole.

Testing a workflow before it is merged

A release workflow only does its real work on a tag, which is a bad moment to find a typo. Two things catch mistakes earlier:

- Every pull request runs it, so a change that breaks the Windows or the Mac build shows on its pull request. Only the `release` job is skipped, since a pull request is not a tag.
- [actionlint](#) checks workflow files, and, with [ShellCheck](#), the shell scripts inside them:

```
pip install actionlint-py shellcheck-py
actionlint .github/workflows/release.yml
```

`concurrency` keeps one run per branch: when a pull request gets a new push, the older run is cancelled, since the newer one contains it. Releases are never cancelled, neither for a tag pushed nor for a run by hand with a tag:

```
concurrency:
  group: release-${{ github.ref }}-${{ inputs.tag }}
  cancel-in-progress: ${{ github.ref_type != 'tag' && !inputs.tag }}
```

The tag is part of the group's name, so a push to main while a release is being built from main starts a run of its own instead of waiting behind it. `!inputs.tag` is true when the field is empty.

Other ways in: the AUR and Flathub

Two more recipes live in `packaging/`, for package managers that build the program themselves from the source:

- `packaging/aur/` holds two **PKGBUILDs** for Arch Linux's user repository: `ferrous-lineup`, built from a release's tag, and `ferrous-lineup-git`, built from main.
- `packaging/flathub/` holds a **Flatpak** manifest for Flathub, with `cargo-sources.json`, every crate of `Cargo.lock` by URL and checksum: Flathub builds without a network, so every download must be listed in advance. `generate-sources.py` writes that file again after `Cargo.lock` changes.

Both install the same desktop entry, icon, MIME type and AppStream file as the AppImage and the `.deb`. Their READMEs say how to publish them.

What the packages don't do yet

- The programs are not fully **signed**. The macOS app has an ad-hoc signature, which Apple Silicon requires, but is not **notarized**: that needs a paid Apple Developer ID, so Gatekeeper asks the first time. The Windows programs are unsigned, so SmartScreen warns. The manual's [Installing it](#) shows how to get past each warning.

How the books are built

Files: `.github/workflows/book.yml`, `book/book.toml`, `book/print.css`, `book/print-pdf.py` and `book/index-words.txt`

Four books live in the repository, next to the code they describe:

Folder	Book	On the web
<code>book/</code>	this book	https://naelolaiz.github.io/ferrous-lineup/
<code>book-es/</code>	this book in Spanish	https://naelolaiz.github.io/ferrous-lineup/es/
<code>manual/</code>	the user manual	https://naelolaiz.github.io/ferrous-lineup/manual/
<code>manual-es/</code>	the user manual in Spanish	https://naelolaiz.github.io/ferrous-lineup/es/manual/

Each is a website and a PDF with contents, page numbers, an index and bookmarks. A change to the app and the pages that describe it go in the same pull request, or in a book pull request right after it, so the books never describe a program that no longer exists.

mdBook

`mdBook` turns a folder of Markdown files into a website. `book.toml` holds the settings, and `src/SUMMARY.md` the order of the chapters, which becomes the sidebar:

```
# Summary

[Introduction](introduction.md)

[Building it from source](building.md)

# Part I: The shape of the program
- [Five crates, and what each one may know](program/crates.md)
```

A link on a line of its own, before the first part, is a chapter without a number, like the introduction. A line starting with `#` is a part's title, and each `- [...] (...)` a numbered chapter (an indented one would be a chapter inside the one above).

`mdbook build` writes the website into the book's `book/` folder, and `mdbook serve --`

`open` builds it, opens it, and builds it again each time a page is saved: the way to read a page before pushing it.

`site-url` in `book.toml` is the path GitHub Pages serves the book from, `/ferrous-lineup/` (or `/ferrous-lineup/es/`). Most pages find their styles with relative links, but the “page not found” page can be shown at any address, so it needs to know where the book starts.

A page that moves keeps its old address working. `book.toml` lists the old paths, each with where its page went, and mdBook writes a small page at the old path that sends the browser on:

```
[output.html.redirect]
"/step-9/band.html" = "../band/band.html"
```

The workflow

The **Book** workflow has two jobs. **build** runs on every push to main and every pull request that changes a book, a manual or the workflow itself; **deploy** runs only after a push to main, or when started by hand (**Actions > Book > Run workflow**).

```
pull_request:
  paths:
    - "book/**"
    - "book-es/**"
    - "manual/**"
    - "manual-es/**"
    - ".github/workflows/book.yml"
```

`paths` is why a change to the app’s code doesn’t build the books, and a book page doesn’t wait for the app’s tests. `pull_request` names no branch, so a book page stacked on another pull request is built too.

1. **mdBook**, pinned to one version, is downloaded ready-built by `taiki-e/install-action`, which checks its checksum. The same version on every run builds the same books, however mdBook changes later.
2. **The websites:** `mdbook build` in each of the four folders.
3. **Fonts.** The books write rests and notes as music symbols (♯, ♭) and have a few emoji. A font without them prints an empty box, so the job installs FreeSerif and Noto Color Emoji.
4. **pypdf**, a Python library for PDF files, pinned too. Ubuntu’s Python won’t install packages for the whole system, so it goes into a **virtual environment**: a folder

with a Python and packages of its own.

5. **The PDFs.** mdBook also writes `print.html`, every chapter on one page, which is what the printer button on the website opens. Each book is built a second time, into `pdf/`, in the light theme, since the website is dark but paper is white. mdBook reads a setting from an environment variable named after it, so `MDBOOK_OUTPUT__HTML__DEFAULT_THEME=light` overrides `default-theme` under `[output.html]` for that build only (`__` between the levels). Then `print-pdf.py` prints that page to a PDF.
6. **Uploads.** Each PDF is a download of the run, pull requests included. The four websites, with their PDFs, are put together into one site: the English book at the root, the Spanish one under `es/`, and each manual under `manual/` in its language.
7. **Publishing.** `deploy-pages` hands the site to GitHub Pages. That job alone asks for `pages: write` and `id-token: write`, to publish and to prove which workflow is publishing; every other job only reads the code. One deployment runs at a time (`concurrency: group: pages`); a newer one waits rather than cancel one halfway.

Finding your way in the PDF

The print page, printed as it is, is hundreds of pages with no page numbers, no contents and no index. On the website the sidebar and the search box find a chapter; on paper nothing would. `print-pdf.py` adds what a printed book needs:

What	Made from
A title page and the contents , every chapter with the page it starts on	<code>SUMMARY.md</code>
A page number at the bottom of every page	<code>print.css</code>
An index at the back: words, and the pages that explain them	the <code>music</code> and <code>Rust</code> glossaries, and <code>index-words.txt</code>
Bookmarks , the outline a PDF reader shows beside the pages	the parts, the chapters and their sections

Page numbers: CSS for paper

When a browser prints, it pours the page onto sheets of paper, and the `@page` rule says what those sheets are:

```
@page {
    size: A4;
    margin: 18mm 16mm;

    @bottom-center {
        content: counter(page);
        font-family: "Open Sans", sans-serif;
        font-size: 9pt;
        color: #555;
    }
}
```

Inside it, *margin boxes* name the spaces around the text: `@bottom-center` is the middle of the bottom margin, and `counter(page)` the number of the page being printed. The title page has none: the script adds a rule for the first page only, `@page :first`.

The contents: printing twice

The contents must say on which page each chapter starts before the book is printed, but only printing decides where a chapter lands: the length of every page before it depends on the fonts, the pictures and where each line wraps. The script does what LaTeX does: it prints twice.

1. The first time, every page number in the contents and the index is `0000`.
2. The script reads in that PDF where each chapter and each word of the index landed.
3. The second time, the real numbers take the blanks' place. `0000` is as wide as any page number of the book, so no line grows, nothing moves, and the numbers are right. The script reads that PDF too, and if something did move, it prints again, up to four times.

Where a chapter landed is read through its links. mdBook gives every heading an `id` and a link to itself, and when Chrome prints a page with a link to `#the-chord-sheet`, it writes a **named destination** into the PDF: the name, the page, and where on the page. pypdf reads them back:

```
def destinations(reader: PdfReader) -> dict[str, tuple[int, object]]:
    found = {}
    for name, dest in reader.named_destinations.items():
        id = urllib.parse.unquote(name.lstrip("/"))
        found[id] = (reader.get_destination_page_number(dest), dest)
    return found
```

The names come back as `/the-chord-sheet`, with every letter outside ASCII written as % codes (the `ö` of Kröpfel is `%C3%B6`), so `unquote` turns them back into the `ids` of the HTML. Chrome only writes a destination for an `id` some link goes to. Every heading links to itself, and the contents and the index link to what they list; when the index shows one number for two places on the same page, a hidden list of links (`<div hidden>`) still points at both.

The index

The books already have lists of words: their glossaries, tables with the word in bold, what it means, and an “Explained in” link. The script reads them from `print.html` and gives each word:

- the page its “Explained in” link goes to, in **bold**;
- every page where a chapter writes the word in bold, which is how these books introduce a word. Each such bold word gets an `id`, such as `pdf-ix-17`, for the index to link to.

A plural counts too, and a word that is all code only matches code: the glossary’s `windows`, a Rust method, is not the bold Windows of “Linux, macOS and Windows”. `index-words.txt` adds the words no glossary has, mostly the parts of the app, with the chapters that explain them:

```
Chord sheet: band/chord-sheet.md, ...
Modulation: see Key change
Function: not program/sync.md
```

- A place is a chapter, by the file `SUMMARY.md` links, or a section of it, by its heading’s `id` after `#`.
- `see` sends the reader to another entry.
- `not` leaves out a chapter where the bold word means something else: there “function” is a Rust function, not what a chord does in a key.

A line that names a chapter or a section the book doesn’t have stops the script with an error, so a renamed heading fails the workflow instead of sending readers to the wrong page.

Bookmarks and links

Chrome writes no outline for this page, so the script adds one with `pypdf` after the last printing: the parts at the top, each chapter under its part, and a chapter’s `##` sections

under the chapter.

```
writer = PdfWriter(clone_from=reader)
add_bookmarks(writer, chapters, dests)
point_links_at_pages(writer)
writer.page_mode = "/UseOutlines"
```

`/UseOutlines` asks the reader to open with the bookmarks shown.

`point_links_at_pages` fixes the links themselves. Chrome writes a link as “go to the place named `the-chord-sheet`”, and a table at the back of the PDF gives each name a place. Chrome’s and Firefox’s readers look the name up, but not every reader does, and in those such a link does nothing. So the script looks every name up itself and writes the place into the link, “go to page 543, this far down”, which every reader follows:

```
link[NameObject("/Dest")] = ArrayObject(place)
```

One script per language

The Spanish book has its own `print-pdf.py`, a copy with the Spanish words: “Contenido” and “Índice alfabético” for the headings, `véase` and `no` in `index-words.txt`, the Spanish glossaries, and two rules English does not need. A word ending in `-ión` has its plural in `-iones` (*inversión*, *inversiones*), and accents don’t move a word in the index: “Índice” sorts with the I’s.

The manuals have no script of their own. Each is printed by its language’s script, given the manual’s folder as a third argument, where the script finds the manual’s `book.toml` and `index-words.txt`.

Printing a PDF yourself

Nothing here needs GitHub. To print the book’s PDF as the workflow does, you need `mdBook`, Python 3.11 or newer, and Chrome or Chromium:

```
python3 -m venv ~/pdf-tools
~/pdf-tools/bin/pip install pypdf
cd book
MDBOOK_OUTPUT__HTML__DEFAULT_THEME=light \
  MDBOOK_OUTPUT__HTML__PREFERRED_DARK_THEME=light \
  mdbook build --dest-dir pdf
~/pdf-tools/bin/python print-pdf.py pdf book/ferrous-lineup-book.pdf
```

If Chrome is not called `google-chrome` on your system, say where it is:

```
CHROME=chromium ~/pdf-tools/bin/python print-pdf.py ...
```

The app is Rust; the books are built with whatever does each job most simply: Markdown for the pages, YAML for the workflow, CSS for the paper, Python and pypdf for the PDF.

Project file format

A Ferrous Lineup project is a text file in **RON**, “Rusty Object Notation”, which looks like Rust’s own struct and enum syntax. The app writes it through `serde`, reads it through the same types, and saves it with the extension `.lineup`.

```
(
  format: 29,
  tempo: 130.0,
  scale: (root: C, kind: Major),
  channels: [
    (name: "Kick", kind: Sampler(Builtin(Kick)), volume: 0.8, muted:
false),
    // ...
  ],
  // ...
)
```

[Saving](#) explains RON, `serde` and the `ProjectFile` struct that stands between the file and the live `Project`. This page is the reference for the number at the top:

`FORMAT_VERSION` is **29** in [crates/lineup-core/src/file.rs](#).

The format number

The number says which features a file may contain. It is not the app’s version.

`format_for` writes the **lowest** format that can hold the project, not always 29. A plain project stays format 1, so any version of Ferrous Lineup can open it. A song with a chord sheet but no band is format 12; a band is format 13; a locked channel is at least format 23. The number is that of the highest feature the song uses, and each format can hold everything the ones below it hold.

Every field a format adds has a `serde` default. A field at its default is left out when writing, and a missing field gets its default when reading. So a song that uses nothing new is written exactly as before, and a file needs no conversion step when it is opened: it carries only what it uses.

Format	What it adds	What a version that reads only the formats below would get wrong	Explained in
1	The base: channels, patterns, tracks, clips, tempo and key.	Nothing; every version reads it.	Saving
2	Channels that play an external MIDI synth.	It would not know the <code>MidiOut</code> channel and would not send the notes out.	MIDI
3	Nine more built-in drums, patterns longer than four bars, and time signatures.	It would not know the drums, would cut long patterns short without a word, and would draw every bar in four.	Tempo, meter, swing and feel
4	Sections, clips that repeat, and a loop region.	It would play every repeat once and lose the loop.	The project
5	Time markers, which change the time signature partway through a song.	It would draw every bar in the song's first time signature.	Tempo, meter, swing and feel
6	A twelve-tone row or a pitch-class set as the key.	It would not know either kind of key and could not open the file.	Twelve-tone rows, Pitch-class sets
7	Tempo changes through a song, and loose timing.	It would play the whole song at its first tempo and put every note on the grid.	Tempo, meter, swing and feel
8	Key changes through a song.	It would show and play the whole song in its first key.	Scales, modes and keys
9	A General MIDI instrument on a channel.	It would play every Lineup Osc with the same sound and	SoundFonts

Format	What it adds	What a version that reads only the formats below would get wrong	Explained in
		send no program changes.	
10	Notes that start or end between steps, such as an imported MIDI file's triplets.	It would move them onto the grid and change the rhythm.	MIDI
11	Beats grouped irregularly, such as 7/8 counted 2+2+3.	It would read 7/8 but draw a beat line on every eighth note.	Tempo, meter, swing and feel
12	The chord sheet: the song's harmony, bar by bar.	It would lose the sheet without a word when saving.	The chord sheet
13	The band that plays the chord sheet, and the patterns it writes.	It would keep the band's patterns as plain patterns and stop following the sheet.	How the band plays
14	The band's song form: an intro, repeated choruses and an ending.	It would play the song once, as written, with no repeats or ending.	Song form
15	The style's second variation, chosen by markers on the chord sheet.	It would play the whole song in the first variation.	Song form
16	Swing, the eighth notes played long-short.	It would play every pair of eighths evenly.	Tempo, meter, swing and feel
17	The band's guitar, strings and organ.	It would know only piano, bass and drums, and could not read their parts.	How the band plays
18	Shots, holds, rests and pushes on the chord sheet.	It would drop the marks and play the	Shots, holds, rests and pushes

Format	What it adds	What a version that reads only the formats below would get wrong	Explained in
		style straight through.	
19	The band's soloist.	It would know no soloist player or soloist part.	The soloist
20	A band's rhythm taken from one style and its instruments from another, the exact harmony style, and accents alone.	It would use one style for every part and lose the accents-only setting.	How the band plays, The orchestra
21	The master mixer, solo and stereo reverb.	It would play the song with the default master and reverb, not the mix you saved.	The mixer
22	Your own styles, saved with the band.	It would fall back to the style it knows by name, so the band would play something else.	Your own styles
23	Locked channels.	It would let styles and the band write over the locked part.	Song form
24	A different style for part of a song.	It would play the whole song in the band's own style.	Song form
25	Two chords at once on the chord sheet, written Eb7 over Fb.	It could not read a polychord, so it could not open the file.	Hearing harmony
26	A pan and a reverb send on each channel.	It would play every channel in the middle with all of it in the reverb.	The mixer

Format	What it adds	What a version that reads only the formats below would get wrong	Explained in
27	Sending a built-in drum over MIDI.	It would play the drum itself instead of sending it to the external synth.	MIDI
28	A horn section in the band, and EQ, compression and delay on a channel.	It would play no horns, and every channel without its effects.	How the band plays, The mixer
29	A band playing the drums of another style than its chord and bass parts, or leaving the drums to the song.	It would play the drums of the band's own style.	How the band plays, The library

Two records can be in a file of any format, because a version that skips them still plays the song exactly. One says where each part of the song came from ([The library](#)). The other is a key written the other way round, such as G $\flat$ major rather than F $\sharp$ major: the key is saved as `(root: FSharp, kind: Major, respelled: true)`, and a version that does not know `respelled` reads it as F $\sharp$ major, with the same notes ([Scales, modes and keys](#)).

Files from a newer version

A newer Ferrous Lineup may write a format this one has never seen. Such a file may contain names and fields this version cannot read, and even if it reads, their meaning may have changed. So the reader checks the format number and refuses anything above `FORMAT_VERSION :`

```
this file uses format 30, made by a newer version of Ferrous Lineup (this
one reads up to format 29)
```

The whole file is read first, and the number checked after. When the text cannot be read at all, the reader looks for the number at its start without reading the rest, so a file from a newer version is explained by its format number rather than by a puzzling unknown field ([Saving](#)).

Saving safely

`file::save` writes to a temporary file next to the target, ending in `.lineup.tmp`, then renames it. A crash or a full disk therefore leaves the old project as it was, and a half-written file is never mistaken for a saved one; if the save fails, the temporary file is removed. Line endings are always `\n`, whatever the system, and a byte-order mark at the start of a file is ignored when reading ([Saving without losing the old file](#)).

Rust glossary

Short definitions of the Rust terms used in this book, with the chapter that explains each one.

Term	Meaning	Explained in
Anonymous lifetime (<code>'_</code>)	A lifetime written without a name, meaning “the lifetime of the borrowed argument”.	Tempo, meter, swing and feel
<code>Arc<T></code>	A shared, reference-counted pointer that is safe to use across threads; the data is freed when the last <code>Arc</code> goes.	Instruments
Array	A fixed number of values stored inline, like <code>[u8; 12]</code> ; the length is part of the type.	Twelve-tone rows and their rules
Array <code>map</code>	<code>[T; N]::map</code> calls a closure on each item of an array and gives back a new array of the same length, with no <code>Vec</code> and no <code>collect</code> .	Twelve-tone rows and their rules
<code>array::from_fn</code>	Builds an array by calling a closure once for each element.	Instruments
<code>as cast</code>	Converts between number types, or a fieldless enum to its number: <code>PitchClass::D as u8</code> is 2.	Five crates, and what each one may know
Associated function	A function in an <code>impl</code> block without <code>self</code> , called as <code>Type::name()</code> . <code>Note::new</code> is one.	Five crates, and what each one may know
Associated type	A type that each implementation of a trait chooses, like <code>type Error = FileError;</code> in an <code>impl TryFrom</code> .	Saving
Atomic	A value (like <code>AtomicUsize</code>) that threads can read and write without a lock.	Two threads

Term	Meaning	Explained in
AtomicU64 for an f64	There is no atomic float, but <code>f64::to_bits</code> turns a float into the <code>u64</code> with the same bits, and <code>f64::from_bits</code> turns it back, so an <code>AtomicU64</code> can share one.	MIDI
Backtracking	Building an answer a piece at a time, and taking the last piece back when it leads nowhere; usually a function that calls itself.	Twelve-tone rows and their rules
Big-endian / to_be_bytes	Writing a number's most significant byte first; <code>6u32.to_be_bytes()</code> is <code>[0, 0, 0, 6]</code> . MIDI files use it.	MIDI
Bit operations	<code>&</code> (and), <code> </code> (or), <code>!</code> (not), <code><<</code> and <code>>></code> (shift) work on the bits of an integer.	Pitch classes, notes and spelling
bool::then / then_some	Turn a condition into an <code>Option</code> : <code>cond.then_some(x)</code> is <code>Some(x)</code> if <code>cond</code> is true and <code>None</code> if not; <code>cond.then(...)</code> only runs the closure when it is true.	Editing tools as plain functions
Borrow	Using a value through a reference (<code>&T</code> shared, <code>&mut T</code> mutable) without taking ownership.	Two threads
Borrow checker	The part of the compiler that enforces: one <code>&mut</code> or any number of <code>&</code> at a time, and no reference outliving its value.	Telling the audio thread what changed
Borrow trait	A promise that an owned type (like <code>PathBuf</code>) hashes and compares exactly like the type it holds (<code>Path</code>), so a <code>HashMap<PathBuf, _></code> can be searched with a <code>&Path</code> .	Telling the audio thread what changed
Box<T>	An owned pointer to a value on the heap.	Drawing the windows with egui
Byte string (<code>b"..."</code>)	A string literal that is a <code>&[u8; N]</code> of ASCII codes instead of a <code>&str</code> , like <code>b"MThd"</code> .	MIDI

Term	Meaning	Explained in
Capacity	How many items a <code>Vec</code> can hold before it must allocate more memory.	Two threads
Cargo	Rust's build tool and package manager.	Five crates, and what each one may know
Cell<T>	A box whose value can be replaced through a shared reference, by moving whole values in and out with <code>get</code> and <code>set</code> (for <code>Copy</code> types); nothing can hold a reference to the inside, so nothing reads it while it changes.	Roman numerals, the harmony keys and chord functions
cfg / cfg_attr	Conditional compilation: <code>#[cfg(target_os = "windows")]</code> keeps the next item only when the condition is true, and <code>cfg_attr</code> applies an attribute only then.	Three systems and continuous integration
#[cfg(test)] on an item	Keeps one function (or <code>use</code> , or struct) only in test builds, so a helper that only the tests call does not warn as unused in a normal build.	Micromodes
checked_add_signed	Adds a signed number to an unsigned one, like <code>usize + isize</code> , and gives <code>None</code> instead of going below zero or overflowing.	Editing tools as plain functions
chunk_by	Splits a slice into runs of neighbours that belong together, without copying.	The editors
chunks / chunks_exact	Split a slice into pieces of <code>n</code> items; <code>chunks_exact</code> leaves out a last piece that is too short.	Twelve-tone rows and their rules
chars / Chars::as_str	<code>text.chars()</code> walks a string one whole character at a time (" <code>b</code> " is three bytes but one <code>char</code>); <code>as_str()</code> is the part not walked	Pitch classes, notes and spelling

Term	Meaning	Explained in
	yet, borrowed from the same text.	
Closure	An anonymous function, <code> args body</code> , that can capture variables from its surroundings.	Two threads
Closure captures fields	Since the 2021 edition, a closure that uses <code>self.plays</code> borrows only that field, not all of <code>self</code> , so other fields can be borrowed at the same time.	The sequencer
<code>cmp::Reverse</code>	Wraps a value so that it compares the other way round: sorting by <code>Reverse(count)</code> puts the largest count first.	Micromodes
Collecting Option s	Collecting an iterator of <code>Option<T></code> into an <code>Option<Vec<T>></code> gives <code>Some</code> of all the values, or <code>None</code> as soon as one item is <code>None</code> . <code>Result</code> works the same way.	Editing tools as plain functions
<code>const fn</code>	A function simple enough to run while compiling, so it can be used to set a <code>const</code> .	Saving
Constant promotion	Writing <code>&[1, 2, 3]</code> of plain constant values gives a <code>&'static</code> borrow, because the compiler stores the values with the program. A function call inside the brackets, even a <code>const fn</code> , is not promoted.	The library
<code>Copy</code>	A marker trait for small values that are copied instead of moved, like numbers and <code>Note</code> ; <code>let held = list[i];</code> then copies the item out.	Five crates, and what each one may know
<code>count_ones</code>	How many bits of an integer are 1: the size of a set stored as bits.	Pitch classes, notes and spelling
Crate	One library or program; the unit Cargo builds and publishes.	Five crates, and what

Term	Meaning	Explained in
		each one may know
cycle	Repeats an iterator forever. Safe because iterators are lazy: something else, like <code>zip</code> , decides when to stop.	Editing tools as plain functions
Declarative macro (<code>macro_rules!</code>)	A macro defined by patterns: <code>macro_rules! hit { (\$at:expr, ...) => { ... } }</code> matches what it is given and pastes it into a template, before the code is compiled.	The library
Default	A trait for “the default value” of a type; on an enum, <code>#[derive(Default)]</code> needs <code>#[default]</code> on one variant.	Five crates, and what each one may know
Deref coercion	Rust turning <code>&Vec<T></code> into <code>&[T]</code> (or <code>&String</code> into <code>&str</code>) automatically where needed.	MIDI
Dereference (<code>*</code>)	Follow a reference to the value it points at, to read or write it.	The project
Derive	<code>#[derive(Trait)]</code> asks the compiler to implement common traits automatically.	Five crates, and what each one may know
Derive macro	Code that writes a trait implementation for a type; <code>serde</code> ’s <code>#[derive(Serialize)]</code> comes from a crate, not from the compiler.	Saving
Display	The trait behind <code>{}</code> in <code>format!</code> and behind <code>to_string()</code> . Implement its <code>fmt</code> method once, and every place writes the value the same way.	Five crates, and what each one may know
Disjoint borrows	Borrowing two different fields of a struct at once, even one mutably; allowed because they cannot overlap.	Drawing the windows with <code>egui</code>

Term	Meaning	Explained in
<code>div_ceil</code>	Integer division that rounds up: <code>16.div_ceil(6)</code> is 3, where <code>16 / 6</code> is 2.	Micromodes
<code>drain(...)</code>	Takes every item out of a <code>Vec</code> , in order, and keeps the <code>Vec</code> 's memory for later.	MIDI
<code>dyn Trait</code>	A trait object: some type implementing <code>Trait</code> , chosen at run time.	Saving
Edition	A version of the language (2015, 2018, 2021, 2024) chosen per crate.	Five crates, and what each one may know
Enum	A type whose value is one of several variants, each of which can hold data.	Two threads
Enum size	An enum takes as much room as its biggest variant; putting a big variant's data in a <code>Box</code> keeps every value small. Clippy's <code>large_enum_variant</code> lint points it out.	MIDI
<code>env!</code> and <code>concat!</code>	Macros that run while the program is compiled: <code>env!("CARGO_PKG_VERSION")</code> becomes the version Cargo read from <code>Cargo.toml</code> , and <code>concat!</code> joins literals into one <code>&'static str</code> , so both can build a <code>const</code> .	Drawing the windows with egui
<code>expect</code>	Like <code>unwrap</code> , but panics with the message given. It states a promise ("this is always <code>Some</code> "), best kept by a test.	Pitch-class sets, and one harmony with four names
Feature	An optional part of a crate, turned on or off in <code>Cargo.toml</code> ; features can only add, never take away.	Saving
<code>find_map</code>	Calls a closure on each item and returns the first <code>Some</code> it gives.	Chords

Term	Meaning	Explained in
<code>flat_map</code>	Maps each item to an iterator and joins all of them into one iterator.	The editors
Float to integer <code>as</code>	<code>x as usize</code> drops the fraction and <i>saturates</i> : negative numbers and NaN become 0, numbers too big become the type's maximum. It never panics.	MIDI
<code>fn</code> pointer	The type of a plain function, like <code>fn(usize) -> Change</code> . Enum variants that hold data are such functions too: <code>Change::RemovePattern</code> builds a <code>Change</code> from a <code>usize</code> .	Telling the audio thread what changed
<code>Fn / FnMut / FnOnce</code>	The traits closures implement: only reads, mutates its captures, or can be called once.	Instruments
<code>From / Into</code>	Conversion traits: implementing <code>From<A> for B</code> also gives <code>a.into()</code> . The <code>?</code> operator uses <code>From</code> to convert errors.	The project
<code>FromIterator</code>	The trait behind <code>collect()</code> ; implement it to collect into your own type.	Pitch classes, notes and spelling
<code>FromStr / str::parse</code>	The trait for “can be read from text”: implement <code>from_str</code> , and <code>"Am7".parse:::<ChordSymbol>()</code> works, as it does for numbers.	Chords
Generic	Code written for a placeholder type <code>T</code> , filled in at compile time.	Two threads
Glob import	<code>use super::*;</code> brings in every name the other module can see, including a parent's private items when used from a child. Two glob imports that bring the same name make it ambiguous.	How a demo is written
<code>HashMap<K, V></code>	A map from keys to values that finds a key in about the same time however many entries it holds.	Telling the audio thread what changed

Term	Meaning	Explained in
<code>if let</code> / <code>while let</code>	Run a block if (or while) a value matches one pattern.	Drawing the windows with egui
<code>ilog2</code>	The integer base-2 logarithm: how many times a number can be halved before reaching 1. <code>8u8.ilog2()</code> is 3. It panics on 0.	Tempo, meter, swing and feel
<code>impl Trait</code>	As an argument type: any type with this trait. As a return type: one type with this trait, left unnamed.	The project
Inner attribute (<code>#![...]</code>)	An attribute that applies to the thing it is written inside, such as the whole crate at the top of <code>main.rs</code> .	Three systems and continuous integration
Invariant	A rule that is always true about a value, usually protected by private fields.	Twelve-tone rows and their rules
<code>is_some_and</code>	<code>true</code> if an <code>Option</code> is <code>Some</code> and its value passes a test, <code>false</code> for <code>None</code> .	Roman numerals, the harmony keys and chord functions
Iterator	A lazy sequence of values, transformed with methods like <code>map</code> , <code>filter</code> , <code>count</code> .	How the code is tested
<code>JoinHandle<T></code>	What <code>thread::spawn</code> returns. <code>join()</code> waits for the thread to end and gives back what its closure returned, or an <code>Err</code> if it panicked.	Exporting
Lazy	Iterators do no work until something pulls items out of them.	Editing tools as plain functions
Leading underscore (<code>_name</code>)	Tells the compiler that a variable or field is unused on purpose, so it does not warn about it.	Exporting
<code>let ... else</code>	Destructure a pattern, or run an <code>else</code> block that must leave the	Telling the audio thread

Term	Meaning	Explained in
	function.	what changed
Let chain	<code>if cond && let Some(x) = value { ... }</code> : conditions and pattern matches in one <code>if</code> (Rust 1.88).	Two threads
Literal in a pattern	<code>Event::Key { repeat: false, .. }</code> : a value written in a pattern only matches when the field equals it.	MIDI
match	Compare a value against patterns; must cover every case.	Two threads
Match guard	<code>pattern if condition =></code> : the arm matches only if the condition is also true.	Chords
matches!	A macro that is <code>true</code> if a value matches a pattern.	Pitch classes, notes and spelling
mem::replace	Put a new value into a place and get the old one back.	Two threads
mem::swap	Exchanges the values behind two <code>&mut</code> references, without copying either.	Undo and redo
mem::take	Take a value out of a place, leaving the type's default (such as an empty <code>Vec</code>) behind.	The sequencer
Method	A function in an <code>impl</code> block that takes <code>self</code> , <code>&self</code> or <code>&mut self</code> .	Five crates, and what each one may know
min_by_key	The item of an iterator with the smallest key, or <code>None</code> if there are no items.	Pitch-class sets, and one harmony with four names
Module	A namespace inside a crate, usually one file, declared with <code>mod</code> .	Five crates, and what each one may know

Term	Meaning	Explained in
Monomorphization	The compiler making a specialized copy of generic code for each type used.	Two threads
Move	Transferring ownership of a value; the old name becomes unusable.	Two threads
<code>move</code> closure	A closure that takes ownership of everything it uses instead of borrowing it; needed for a closure given to another thread.	Exporting
<code>mpsc</code> channel	A queue from the standard library with many senders (<code>Sender</code> can be cloned) and one receiver; <code>try_iter</code> takes what is waiting without waiting for more.	MIDI
<code>Mutex<T></code> and poisoning	A lock that lets one thread at a time reach the <code>T</code> inside. If a thread panics while holding it, the mutex is <i>poisoned</i> , and <code>lock</code> returns an error until you take the data anyway with <code>PoisonError::into_inner</code> .	MIDI
Newtype	A struct with one field that wraps another type to give it its own meaning.	Pitch classes, notes and spelling
<code>?</code> operator	Return early with the error if a <code>Result</code> is <code>Err</code> (or an <code>Option</code> is <code>None</code>).	Two threads
<code>Option::and_then</code>	Calls a closure that itself returns an <code>Option</code> on the value, if there is one; <code>None</code> stays <code>None</code> .	The editors
<code>Option::as_mut</code>	Turns <code>&mut Option<T></code> into <code>Option<&mut T></code> , to use the value inside without moving it out.	Two threads
<code>Option::filter</code>	Keeps a <code>Some</code> only if a closure says yes, and gives <code>None</code> otherwise.	MIDI
<code>Option<T></code>	<code>Some(value)</code> or <code>None</code> ; Rust's replacement for null.	Two threads

Term	Meaning	Explained in
Option::take	Moves the value out of an <code>Option</code> and leaves <code>None</code> in its place; handy to take ownership of something stored in a field.	Exporting
Or-pattern	<code>A B</code> in a pattern matches either <code>A</code> or <code>B</code> .	Drawing the windows with egui
Ownership	Every value has exactly one owner, and is dropped when the owner goes away.	Two threads
partition	Splits an iterator in two by a closure, in one pass: the items it says yes to and the rest, each collected into its own collection.	Micromodes
partition_point	On a slice sorted so that a test is <code>true</code> for a first part and <code>false</code> for the rest, the index where the <code>false</code> part begins, found by binary search.	Pitch-class sets, and one harmony with four names
Panic	An unrecoverable error that stops the current thread (<code>panic!</code> , failed <code>assert!</code>).	Five crates, and what each one may know
Property-based test	A test that checks a rule which must hold for any input (“following the changes gives the new project”), rather than one answer for one input.	Telling the audio thread what changed
pub(crate)	Visible everywhere inside the current crate, but not to other crates.	The project
pub(super)	Visible to the parent module and everything inside it.	Saving
Pure function	A function whose result depends only on its arguments and that changes nothing else; the easiest kind of code to test.	Editing tools as plain functions
Rc<T>	A reference-counted pointer for one thread: cloning it adds to a count instead of copying the	Pitch-class sets, and one harmony

Term	Meaning	Explained in
	value, which is freed when the last <code>Rc</code> goes. <code>Rc<T></code> shares a slice.	with four names
RAII / Drop	Owning a value means owning its resource; dropping it releases the resource.	Two threads
Range	<code>a..b</code> , the numbers from <code>a</code> up to but not including <code>b</code> ; a value with methods like <code>contains</code> .	The sequencer
Recursion	A function that calls itself on a smaller piece of the problem, until a piece is small enough to answer directly.	Twelve-tone rows and their rules
rem_euclid	A remainder that is never negative: <code>(-1_i32).rem_euclid(12)</code> is 11, while <code>-1 % 12</code> is -1.	Pitch classes, notes and spelling
Result<T, E>	<code>Ok(value)</code> or <code>Err(error)</code> ; Rust's way to return errors.	Two threads
retain	Keeps only the items of a <code>Vec</code> for which a closure returns <code>true</code> , in one pass.	The project
RON	Rust Object Notation, a text format that looks like Rust's own struct syntax; Ferrous Lineup's project files use it.	Saving
rsplit_once / split_once	Splits a string at the last (or first) place a pattern occurs, into the parts before and after, or <code>None</code> if it does not occur.	Chords
rposition	The index of the last item of a slice that matches, searching from the end; <code>position</code> searches from the start.	Micromodes
saturating_sub	Subtraction that stops at the smallest value (0 for <code>usize</code>) instead of overflowing.	Editing tools as plain functions
Send	A marker trait: values of this type can be moved to another thread.	Two threads

Term	Meaning	Explained in
serde	The usual crate for turning Rust values into text or bytes and back, through the <code>Serialize</code> and <code>Deserialize</code> traits.	Saving
serde container attributes	<code>#[serde(try_from = "String", into = "String")]</code> on a type saves it as the <code>String</code> its <code>From</code> gives, and reads it back through its <code>TryFrom<String></code> , so a value read from a file is checked like any other.	Twelve-tone rows and their rules
serde field attributes	<code>#[serde(default)]</code> fills in a missing field with its <code>Default</code> when reading; <code>#[serde(skip_serializing_if = "path")]</code> leaves a field out when writing if that function says so.	Saving
Slice	A borrowed view of part of an array or <code>Vec: &[T]</code> or <code>&mut [T]</code> .	Two threads
Slice pattern	<code>let [a, b, c, ..] = *bytes</code> else <code>{ ... }</code> : matches a slice by its shape, naming the first items, so no indexing can panic.	MIDI
Slice pattern with @	<code>[first @ .., last]</code> matches a slice of two or more items and names everything but the last as the slice <code>first</code> .	Micromodes
State machine	Code that keeps a current state, often an enum, and moves between states as things happen.	Instruments
'static	A lifetime meaning “lives as long as needed”; for closures, “borrows nothing temporary”.	Two threads
static item	<code>static NAME: T = ...;</code> is one value at one fixed address for the whole program. A <code>const</code> is pasted in wherever it is used instead.	The library
step_by	Takes every n-th item of an iterator: <code>(0..8).step_by(2)</code> is 0,	Editing tools as plain

Term	Meaning	Explained in
	2, 4, 6.	functions
Struct update syntax	<code>NoteEvent { length: 2, ..note }</code> : a new struct with some fields changed and the rest taken from <code>note</code> .	Drawing the windows with egui
Target	The kind of system a program is compiled for, like <code>x86_64-pc-windows-gnu</code> ; <code>rustup target add</code> installs Rust's library for one.	Three systems and continuous integration
<code>thread::Builder</code>	Starts a thread with options such as a name; its <code>spawn</code> returns a <code>Result</code> instead of panicking when the system refuses.	MIDI
<code>thread_local!</code>	Declares a value of which each thread has its own copy, reached with <code>.with(...)</code> ; the app keeps its settings for numerals, note names and language this way, so tests running side by side never see each other's.	Roman numerals, the harmony keys and chord functions
Trait	A set of methods a type promises to provide, like an interface.	Five crates, and what each one may know
Trait bound	A requirement on a generic type, such as <code>T: FromSample<f32></code> .	Two threads
<code>TryFrom / TryInto</code>	Conversions that can fail, returning a <code>Result</code> .	Saving
Tuple	A fixed group of values, possibly of different types, like <code>(usize, Note)</code> ; it can be matched and taken apart.	Pitch classes, notes and spelling
Tuple comparison	Tuples compare item by item: the second item only decides when the first ones are equal. A tuple as a sort key gives tie-breaks in order.	The mixer
Tuple struct	A struct whose fields have no names, like <code>struct Note(u8)</code> ; the fields are read as <code>.0</code> , <code>.1</code> ...	Five crates, and what

Term	Meaning	Explained in
		each one may know
Turbofish (::<>)	Writing a generic function's type out at the call, as in <code>collect::<Vec<Note>>()</code> , when Rust cannot work it out.	Two threads
Type alias	A second name for an existing type: <code>type Hit = (u8, i32);</code> .	The library
VecDeque<T>	A double-ended queue: a ring buffer that is cheap to add to and take from at both ends.	Undo and redo
windows	Every run of <code>n</code> neighbours of a slice, overlapping: <code>[1, 2, 3].windows(2)</code> gives <code>[1, 2]</code> and <code>[2, 3]</code> .	How the code is tested
wrapping_add / wrapping_mul	Arithmetic that wraps around on overflow instead of stopping a debug build, for when wrapping is what you want, as in random numbers.	Twelve-tone rows and their rules
Workspace	A group of crates built together with one <code>Cargo.lock</code> .	Five crates, and what each one may know
zip	Pairs the items of two iterators, <code>(a, b)</code> , and stops when either runs out.	Telling the audio thread what changed

Where to learn more

- *The Rust Programming Language* ("the book"): installed with Rust, open it offline with `rustup doc --book`.
- *Rust by Example*: `rustup doc --rust-by-example`.
- The standard library docs: `rustup doc --std`.

Music glossary

This book uses the words musicians use. This page gives each one a short meaning and the chapter that shows it at work. The [Rust glossary](#) is separate: it explains the language, not the music.

Term	Meaning	Explained in
Comping	The chords a pianist or guitarist plays under the tune, and the rhythm and voicings they use. As a verb, to comp is to play those chords.	How the band plays a chord sheet
Voicing	Which notes of a chord sound, in which octaves, and how they are spread. A good voicing keeps common notes and leaves the melody room.	How the band plays a chord sheet
Voice leading	Moving each voice of a chord to the nearest voice of the next, so the harmony joins smoothly instead of jumping.	Roman numerals, the harmony keys and chord functions
Changes	The chord progression of a song, bar by bar. “The changes” are what the chord sheet holds.	How the band plays a chord sheet
Head	The written tune, played at the start and again after the solos. In a jazz performance, the head is the melody everyone knows.	Song form, sections, variations and locks
Chorus	In jazz, one complete pass through the form, from the top of the chart to the last bar. In a pop song, the repeated part that comes after each verse.	Song form, sections, variations and locks
Turnaround	The chords at the end of a section that lead back to its beginning, such as I–vi–ii–V.	Song form, sections, variations and locks
Walking bass	A bass line of one note per beat: the root, notes of the chord, and a step into the next chord’s root.	How the band plays a chord sheet

Term	Meaning	Explained in
Slash chord	A chord symbol with a bass note after a slash, such as C/E . The bass may be a note of the chord (an inversion) or a note outside it.	Chords
Inversion	A chord with a note other than its root in the bass. In a triad, the first inversion has the third lowest and the second the fifth; in a seventh chord, the third inversion has the seventh.	Chords
Figured bass	A continuo shorthand: numbers above or below the bass note name the intervals to play above it, such as 6, 6/4 or 6/5. Ferrous Lineup writes the figures after the Roman numeral, as I^6 and V^6_5 .	Roman numerals, the harmony keys and chord functions
Roman numeral	A chord named by the degree of the key it is built on: upper case for major, lower case for minor. In C major, G7 is V7.	Roman numerals, the harmony keys and chord functions
Enharmonic keys	Two names for the same keys of the piano, such as F# major (six sharps) and Gb major (six flats). They sound the same; the composer picks the name that fits the music around it.	Scales, modes and keys
Function	What a chord does in the key: tonic is home, subdominant is away, dominant is tension that pulls home. The harmony keys colour them blue, green and orange.	Roman numerals, the harmony keys and chord functions
Borrowed chord	A chord taken from the parallel key, the major or minor on the same tonic, such as A^b (bVI) in C major, borrowed from C minor. Also called modal mixture .	Roman numerals, the harmony keys and chord functions
Picardy third	A major tonic chord at the end of a piece in a minor key: the major third borrowed from the major.	Roman numerals, the harmony keys and chord functions

Term	Meaning	Explained in
Secondary dominant	The dominant of a chord of the key other than the tonic, written V/V ("five of five"), V7/vi and so on. D7 in C major is V7/V, the dominant of G.	Roman numerals, the harmony keys and chord functions
Tonicization	Making a chord of the key sound for a moment like a tonic, by playing its own dominant or leading-tone chord before it, without changing key.	Roman numerals, the harmony keys and chord functions
Cadential six-four	The tonic triad in second inversion over the dominant's bass, such as C/G before G7 in C major. It sounds like the dominant held back, and moves on to V.	Roman numerals, the harmony keys and chord functions
Neapolitan sixth	A major chord on the lowered second degree, usually in first inversion: D $\flat$ /F in C major, written N ^{6} . It mostly goes to the dominant.	Roman numerals, the harmony keys and chord functions
Pivot chord	A chord that belongs to both the old and the new key in a modulation, heard in the first and then reread in the second: Am in C major is vi, and ii in G major.	Roman numerals, the harmony keys and chord functions
Step, skip, leap	How far one part moves: a second is a step, a third a skip, a fourth or more a leap.	Voice leading and part motion
Contrary motion	Two parts moving opposite ways. Between the top line and the bass it keeps them two independent lines.	Voice leading and part motion
Oblique motion	One part holds its note while the other moves.	Voice leading and part motion
Parallel motion	Two parts moving the same way and keeping the same interval number, such as sixths going to sixths. Parallel fifths and octaves are what classical part writing avoids.	Voice leading and part motion
Close position	A chord voiced with its notes as near together as they go, all within an	How the band plays a chord

Term	Meaning	Explained in
	octave, as a horn section plays it.	sheet
Equalizer (EQ)	A filter that turns ranges of frequencies up or down, such as a channel's lows, middle and highs. It changes a sound's colour, not its notes.	The mixer
Compressor	An effect that turns a sound down when it goes over a threshold, by a ratio, so a part plays at a steadier level.	The mixer
Delay	An effect that plays a sound again a little later, and again, each echo quieter; in the app, in time with the song.	The mixer
Polychord	Two chords sounded at once, written one above the other, such as Eb7 over Fb. The ear hears both chords, not one large chord.	Hearing harmony
Pitch-class set	A group of notes named by pitch class, ignoring octave and order. Every chord is one, but the term also covers groups that have no chord name.	Pitch-class sets, and one harmony with four names
Prime form	The most compact transposition or inversion of a pitch-class set, used to compare sets. Forte's catalogue lists each set class with its prime form and interval vector.	Pitch-class sets, and one harmony with four names
Twelve-tone row	An ordering of all twelve pitch classes used as the material of a piece. Its forms are prime, inversion, retrograde and retrograde inversion.	Twelve-tone rows and their rules
Micromode	One of Francisco Kröpfl's twelve three-note groups, classified by interval content and heard as a colour. The app can use one as a scale or draw with it.	Micromodes
Swing	Playing the second eighth note of each beat late, so pairs go long-short.	Tempo, meter, swing and feel

Term	Meaning	Explained in
	A full swing, 100%, is a triplet: two thirds and one third.	
Shot	A short, accented chord hit by the whole band, followed by silence. Also called a stab, a hit or a kick.	Shots, holds, rests and pushes
Push	A chord struck before its beat and held over it, written $\wedge F$; also called an anticipation. It makes the music lean forward.	Shots, holds, rests and pushes
Hold	A chord struck and held until the next harmony, with the beat going on underneath. It is not a fermata.	Shots, holds, rests and pushes
Rest	The band dropping out for a bar or more; in jazz, a break. In the app a rest also leaves the drums silent.	Shots, holds, rests and pushes
Concert pitch	Music written as it sounds, "in C": what piano, guitar, bass, voice, flute and violin read. A song's concert key is the key the band hears.	The lead sheet
Transposing instrument	An instrument whose written notes are not the notes it sounds, named after the note a written C sounds: a trumpet in B $\flat$, an alto saxophone in E $\flat$, a horn in F. Its music is written higher by that distance, so a song in C is in D for a trumpet.	The lead sheet
Solfège	Naming the notes with syllables, Do Re Mi Fa Sol La Si, instead of the letters C D E F G A B, as in Spanish, Italian, French and Portuguese. A chord is named the same way: Am7 is Lam7.	Two languages
Fixed do	Solfège in which Do is always C, whatever the key, as the app and Spanish and Italian schools use. In movable do , used by some ear-training methods, <i>do</i> is the tonic of the key.	Two languages

Ideas for what comes next

None of these are planned yet. They are ideas, roughly from smaller to bigger, and each would make a good project for a reader of this book:

- **More of the band:** more styles, and more instruments it can bring to any of them, as it brings the horns now ([A horn section](#)).
- **A soloist that is no longer an experiment** ([The soloist](#)).
- **Recording in song mode:** today Rec switches to pattern mode and records into the pattern that loops ([exercise 22 of Part II](#)).
- **Audio clips** on the tracks, next to the patterns, placed and moved like any clip. An imported recording now plays as one long sampler note on the first step of the song, which the engine starts partway when playback starts later ([Starting partway](#)).
- **Automation:** knob movements recorded or drawn over time, on the tracks.

Index

The words of the glossaries, the parts of the app and the chapters that explain them. A number in **bold** is the page the glossary sends you to for that word; the others are where a chapter introduces it or is about it.

A

About window, [122](#)
Anonymous lifetime (`'_`), [220](#)
`Arc<T>` , [228](#)
Arpeggio, [477](#)
Array, [521](#)
Array `map` , [521](#)
`array::from_fn` , [228](#)
`as cast`, [18](#)
Associated function, [16](#), [17](#)
Associated type, [78](#), [84](#)
Atomic, [22](#), [32](#)
`AtomicU64` for an `f64` , [285](#)
Audio thread, [22](#), [51](#), [189](#)
Autosave, [95](#)

B

Backtracking, [522](#), [523](#)
Big-endian / `to_be_bytes` , [285](#)
Bit operations, [407](#)
`bool::then` / `then_some` , [468](#)
Borrow, [22](#)
Borrow checker, [54](#)
`Borrow` trait, [51](#), [68](#)
Borrowed chord, [463](#)
`Box<T>` , [98](#)
Byte string (`b"..."`), [285](#)

C

Cadential six-four, [463](#)
Capacity, [22](#), [30](#)
Cargo, [12](#)
`Cell<T>` , [455](#)

`cfg` / `cfg_attr` , [873](#)
`#[cfg(test)]` on an item, [553](#)
Changes, [668](#), [669](#)
Channel rack, [126](#)
`chars` / `Chars::as_str` , [406](#)
`checked_add_signed` , [468](#)
Chord names, [429](#)
Chord recognition, *see* Hearing chords
Chord sheet, [508](#), [633](#)
Chorus, [716](#), [717](#), [833](#)
`chunk_by` , [126](#)
`chunks` / `chunks_exact` , [522](#)

Clip, [36](#), [154](#)
Close position, [692](#)
Closure, [22](#), [28](#), [54](#), [720](#), [752](#), [807](#)
Closure captures fields, [197](#)
`cmp::Reverse` , [555](#)
Collecting `Options`, [468](#)
Comping, [614](#), [668](#), [669](#), [728](#)
Compressor, [272](#), [273](#)
Concert pitch, [798](#), [799](#), [810](#), [811](#)
`const fn` , [78](#)

Constant promotion, [615](#)
Continuous integration, [873](#)
Contrary motion, [564](#)
`Copy` , [17](#)
`count_ones` , [407](#)
Crate, [12](#)
`cycle` , [468](#)

D

Declarative macro (`macro_rules!`), [615](#)
`Default` , [17](#)
Delay, [272](#), [274](#), [389](#)

Demo, [837](#), [853](#)

Deref coercion, [289](#)

Dereference (`*`), [36](#)

Derive, [17](#)

Derive macro, [78](#), [80](#)

Disjoint borrows, [102](#)

`Display`, [17](#)

`div_ceil`, [556](#)

`drain(..)`, [285](#)

Drum kit, [230](#), [278](#), [617](#)

`dyn Trait`, [87](#)

E

Ear training, [770](#)

Edition, [12](#), [15](#)

Enharmonic keys, [417](#)

Enum, [22](#), [23](#)

Enum size, [307](#)

`env!` and `concat!`, [122](#)

Envelope, [241](#)

Equalizer (EQ), [272](#)

`expect`, [505](#)

Export, see WAV export, Video export

F

Feature, [78](#), [81](#), [579](#), [874](#), [875](#)

Figured bass, [446](#), [452](#)

`find_map`, [429](#)

Finding the key, [494](#)

Fixed do, [165](#)

`flat_map`, [126](#)

Float to integer `as`, [285](#)

`Fn` / `FnMut` / `FnOnce`, [231](#)

`fn` pointer, [51](#)

`From` / `Into`, [42](#)

`FromIterator`, [398](#)

`FromStr` / `str::parse`, [439](#)

Function, [449](#), [457](#)

G

Generic, [26](#), [70](#), [688](#)

Glob import, [839](#)

H

Harmony keys, [457](#), [507](#)

`HashMap<K, V>`, [51](#)

Head, [717](#)

Hearing chords, [481](#)

Hold, [737](#), [738](#)

Horn section, [691](#)

I

`if let` / `while let`, [98](#)

`ilog2`, [226](#)

`impl Trait`, [42](#)

Importing a recording, [575](#)

Inner attribute (`#![...]`), [873](#)

Invariant, [518](#)

Inversion, [430](#), [434](#), [438](#), [446](#), [468](#), [785](#), [790](#)

`is_some_and`, [465](#)

Iterator, [868](#)

J

`JoinHandle<T>`, [337](#)

K

Key change, [421](#), [424](#), [496](#)

Key marker, [424](#)

Keyboard shortcuts, [76](#), [121](#), [144](#)

L

Language, [164](#)

Latin American music, [837](#), [853](#)

Lazy, [473](#)

Lead sheet, [798](#)

Leading underscore (`_name`), [337](#)

`let ... else`, [59](#), [60](#)

Let chain, [22](#), [29](#), [61](#), [138](#), [294](#), [420](#), [476](#), [571](#), [643](#), [757](#)

Library, [613](#)

Lineup Osc, [240](#)

Literal in a pattern, [299](#)

Locked track, [731](#)

M

`match`, [22](#)
Match guard, [108](#), [222](#), [288](#), [416](#), [444](#), [494](#),
[641](#), [755](#)
`matches!`, [398](#)
`mem::replace`, [22](#)
`mem::swap`, [69](#)
`mem::take`, [189](#)
Meter, *see* Time signature
Method, [16](#), [17](#)
Micromode, [508](#), [511](#), [543](#), [608](#), [849](#)
MIDI file, [317](#), [323](#), [853](#)
MIDI keyboard, [290](#)
MIDI Out, [307](#), [311](#)
`min_by_key`, [502](#)
Mixer, [265](#)
modal mixture, *see* Borrowed chord
Modulation, *see* Key change
Module, [12](#), [15](#)
Monomorphization, [22](#), [27](#), [71](#)
movable do, *see* Fixed do
Move, [22](#), [24](#)
`move` closure, [337](#)
`mpsc` channel, [285](#)
`Mutex<T>` and poisoning, [285](#)

N

Neapolitan sixth, [463](#)
Newtype, [193](#), [224](#), [401](#), [518](#)
Note naming, [164](#)

O

Oblique motion, [564](#)
Odd meter, [205](#), [700](#)
`? operator`, [22](#)
`Option::and_then`, [141](#)
`Option::as_mut`, [29](#)
`Option::filter`, [285](#)
`Option::take`, [337](#)
`Option<T>`, [22](#), [26](#)
Or-pattern, [99](#), [510](#)

Orchestra, [700](#)
Ownership, [22](#), [24](#), [28](#)

P

Panic, [18](#), [457](#)
Parallel motion, [564](#)
`partition`, [550](#)
`partition_point`, [513](#)
Pattern, [36](#)
Piano roll, [126](#)
Picardy third, [463](#)
Pitch class, [398](#)
Pitch-class set, [488](#), [501](#), [506](#), [507](#), [508](#), [511](#),
[518](#), [529](#), [534](#), [608](#)
Pivot chord, [450](#), [466](#)
Playback trace, [369](#)
Playlist, [36](#), [154](#)
Polychord, [481](#)
Practice, [770](#)
Preferences, [117](#), [257](#)
Prime form, [501](#), [502](#), [544](#)
Project file, [78](#), [899](#)
Property-based test, [62](#)
`pub(crate)`, [41](#)
`pub(super)`, [78](#)
Pure function, [468](#)
Push, [737](#), [739](#), [742](#)

R

`RAII / Drop`, [22](#)
Range, [191](#)
`Rc<T>`, [513](#)
Recording MIDI keys, [301](#)
Recursion, [522](#)
Release packages, [880](#)
`rem_euclid`, [398](#)
Rest, [737](#), [738](#)
`Result<T, E>`, [22](#), [26](#)
`retain`, [39](#)
Roman numeral, [449](#), [450](#)
RON, [78](#), [80](#), [899](#)

`rposition`, [556](#)
`rsplit_once` / `split_once`, [438](#)

S

Sampler, [233](#)
`saturating_sub`, [477](#)
Scale, [412](#)
Secondary dominant, [445](#), [450](#), [463](#)
Section style, [726](#)
`Send`, [27](#), [346](#)
`serde`, [78](#), [80](#)
`serde` container attributes, [518](#)
`serde` field attributes, [80](#)
Shot, [737](#), [738](#)
Slash chord, [438](#)
Slice, [27](#), [654](#), [741](#)
Slice pattern, [285](#), [288](#), [496](#), [554](#)
Slice pattern with `@`, [553](#)
Snap to scale, [427](#)
Solfège, [164](#), [165](#)
Soloist, [746](#)
Song form, [715](#)
Song structure, [45](#)
SoundFont, [246](#)
Speed trainer, [770](#)
State machine, [228](#), [241](#)
`'static`, [27](#), [346](#)
`static` item, [616](#)
Step, skip, leap, [563](#)
`step_by`, [477](#)
Struct update syntax, [99](#), [303](#), [418](#), [469](#), [530](#),
[589](#), [616](#), [842](#)
Style, [616](#), [668](#), [726](#)
Style editor, [760](#)
Swing, [220](#), [328](#), [384](#), [622](#), [728](#), [746](#), [761](#), [765](#),
[824](#), [825](#), [826](#), [830](#)

T

Tango, [837](#), [853](#)
Target, [873](#), [879](#)
Tempo, [202](#)

`thread::Builder`, [285](#)
`thread_local!`, [455](#), [456](#)

Time signature, [202](#)
Tonicization, [463](#)
Track, [154](#)
Trait, [17](#), [809](#)
Trait bound, [22](#), [27](#), [71](#)
Translation, *see* Language
Transport, [198](#), [220](#)
Transposing instrument, [798](#), [810](#)
`TryFrom` / `TryInto`, [78](#)
Tuple, [24](#), [136](#), [400](#), [509](#), [555](#), [755](#)
Tuple comparison, [281](#)
Tuple struct, [16](#), [732](#), [756](#)
Turbofish (`::<>`), [26](#)
Turnaround, [669](#), [717](#)
Twelve-tone row, [516](#), [528](#), [529](#), [533](#), [548](#), [607](#),
[608](#)
Type alias, [617](#)

U

Undo, [69](#)

V

Variations, [721](#)
`VecDeque<T>`, [69](#)
Velocity, [149](#), [848](#)
Version, [122](#), [880](#)
Video export, [337](#)
Voice leading, [459](#), [516](#), [533](#), [563](#), [607](#), [610](#),
[669](#)
Voicing, [668](#), [669](#)

W

Walking bass, [481](#), [668](#), [669](#)
WAV export, [337](#)
`windows`, [868](#)
Workspace, [12](#), [14](#)
`wrapping_add` / `wrapping_mul`, [522](#)

Z

`zip`, [51](#)

Zoom, [107](#)