

Workflow: from a real netlist to a placed diagram, and back

v0.72.77.212

Péter Felecan

assisted by Claude Sonnet 5.5 (Anthropic)

October 12, 2026

Contents

1	Purpose of this document	1
2	The shared pipeline	2
3	Stage 1: getting a real netlist	3
4	Stage 2: classification, shared by both outputs	4
5	Stage 3a: a breadboard-style diagram	5
6	Stage 3b: a new, algorithmically laid-out KiCad schematic	6
7	Stage 4: from build instructions back to a breadboard or a new KiCad schematic	7

1 Purpose of this document

This document describes the actual, concrete, real, command-line path from a real schematic (or a human-readable set of build instructions) to a breadboard-style diagram, a new KiCad schematic, or both – which real command, in which order, produces which real output. CORRECTED 2026-09-16, Péter’s own, direct, repeated correction: every real command below is this project’s own actual, real command-line tool, run exactly as an end user would run it – never a raw Python function call. One real circuit, this project’s own altimeter example (four real components: a Seeeduino XIAO, a BME280 pressure sensor,

a micro SD card module, and a LiPo battery), is used throughout, start to finish, rather than a different circuit per stage.

Terms. The words of this document – zone (the code says sector), board, strip, column, row, rail, anchor, passive, jumper, link, tier, and the schematic words (sheet, net, symbol, unit) – are defined once, in the [Glossary](#) at the end of the concept document, which also settles where this project’s usage differs from the usual breadboard vocabulary (its column is what many makers call a row). Where a term below is unclear, look it up there. The practices the steps below follow, and those the tool does not yet follow, are in the [Best practices](#) section. The real boards the instructions are written for are classified in the [Taxonomy of breadboards](#).

It is a practical, verified walkthrough, not a rationale: every command below was actually run and its real output checked, not described from memory. It does not explain why each stage’s own algorithm works the way it does, or the history of what was tried and found along the way – that narrative lives in ResearchJournal.org’s own dated entries. Read this document to trace a real invocation through the pipeline; read ResearchJournal.org for why each stage is built the way it is.

2 The shared pipeline

Both of this project’s own real outputs – a KiCad schematic and a breadboard-style diagram – start from the exact same place: a real netlist, and the exact same classification step. They diverge only at the final placement stage, which is genuinely different for the two output kinds (a breadboard’s own rows and columns are not a schematic sheet’s own X/Y coordinates), then can converge again: a freshly generated KiCad schematic is, itself, a real `.kicad_sch` file – exactly the same kind of input this whole pipeline started from. A SEPARATE, real, complementary path (Stage 4) starts from human-readable build instructions instead, and reaches the SAME two real destinations.

existing `.kicad_sch`

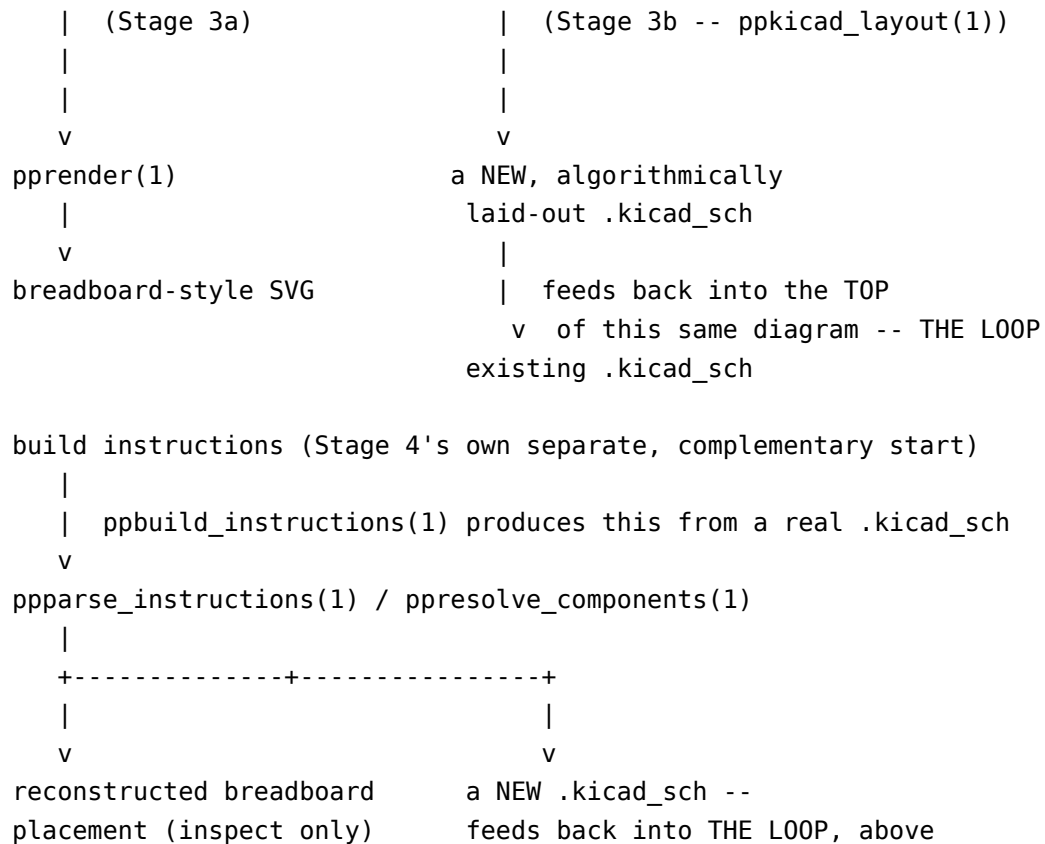
(own output, OR a genuinely third-party design, e.g. TestSchemes/)

```
|
| ppnetlist(1) / pprender(1) / ppauto_layout(1) -- each shells
| out to kicad-cli sch export netlist itself; no separate
| export step needed (see Stage 1)
v
```

a real netlist -- components, nets

```
|
| ppauto_layout(1) -- anchors/passives/rail nets (Stage 2)
v
```

```
+-----+-----+
|               |
| BREADBOARD PATH | KICAD PATH
```



3 Stage 1: getting a real netlist

Two real, equally valid starting points:

- **An existing schematic, already on disk** – either one this project's own tools generated earlier, or a genuinely external design (TestSchemes/ is exactly this: six real KiCad projects, none authored by this project's own tooling – this project's own real altimeter example, TestSchemes/PoorManAltimeter/altimeter.kicad_sch, used throughout this whole document, is one of them). Every real tool below shells out to `kicad-cli sch export netlist` itself – no separate export step needed. Hierarchical, multi-sheet designs (two of TestSchemes' own six projects) are followed automatically by `kicad-cli` itself, provided every real sheet file is still present alongside the root (see ResearchJournal.org's own 2026-08-18 TestSchemes entry for a real, confirmed case of what happens when one isn't: a silent, partial netlist, no warning).
- **A circuit with no schematic yet** – this project's own two, self-contained example circuits (`ppexample_circuit1(1)`, `ppexample_circuit2(1)`) build one directly, with no real .kicad_sch file involved at all. Useful for a circuit still being designed, or for this project's own synthetic stress tests.

Verified, the existing-schematic path, using `ppnetlist(1)` (see README/Index.org's own "Manual pages" section for every real tool this document uses):

```
$ kicad-cli sch export netlist --format kicadsexpr \
  -o altimeter.net TestSchemes/PoorManAltimeter/altimeter.kicad_sch
$ ppnetlist altimeter.net
4 components, 29 nets
BT1: LiPo 3.7V 250mAh
U1: Seeeduino XIAO
U2: BMP280_KK
U3: Micro SD Card Module
GND: BT1.2, U1.13, U2.GND, U3.GND
Net-CS: U1.4, U3.CS
Net-MISO: U1.10, U3.DATA_OUT
Net-MOSI: U1.11, U3.DATA_IN
Net-SCK: U1.9, U3.SCLK
Net-SCL: U1.6, U2.SCK
Net-SDA: U1.5, U2.SDI
VBUS: BT1.1, U1.14, U2.VDD, U3.VDD
unconnected-(U1-3V3_OUT-Pad12): U1.12
...
```

(the trailing `unconnected-(...)` entries, one per real, genuinely unused pin on each real symbol, are trimmed above for length – `ppnetlist` itself prints every one of them).

4 Stage 2: classification, shared by both outputs

`ppauto_layout(1)` (and every other real tool downstream of it) splits every real component into **anchors** (3 or more real net memberships – ICs, connectors, anything with enough real pins to anchor other things around) and **passives** (fewer – resistors, capacitors, diodes, anything that typically attaches to something else rather than anchoring its own), and finds **rail nets** (power/ ground nets with enough real members, or a real, conventional rail name, that drawing them as ordinary point-to-point wires would be unreadable – drawn as a rail/bus symbol instead on the KiCad side, or a rail stub on the breadboard side).

```
$ ppauto_layout TestSchemes/PoorManAltimeter/altimeter.kicad_sch
Anchors (3): ['U1', 'U2', 'U3']
Passives (0): []
Rail nets excluded from placement: []

Anchor placement:
  U1: starts at column 5
```

U2: starts at column 31

U3: starts at column 19

Passive placement:

Every passive was placed.

BT1, this circuit's own real battery, is deliberately absent from BOTH the anchor and the ordinary-passive lists here – not unplaced, not an error. This board's own real, default configuration already has a rail, and a genuine, simple power source is placed directly ON that rail instead of the ordinary main grid at all – see Stage 4's own dedicated section on this, below, for the real, direct account of why and how.

5 Stage 3a: a breadboard-style diagram

`pprender(1)` takes classification one step further: a real row/ column placement (which breadboard column each component's own near leg sits in, which row, whether it needed relocating for room – see FutureResearch.org's own row-assignment item history), verified for connectivity and board-bounds before drawing it.

A single, physical breadboard's own real geometry (sector count, per-sector column/row counts, rail presence and grouping) is described by a config file, not assumed – see `breadboard_config(5)` for the full format, including worked examples; both `ppauto_layout(1)` and `pprender(1)` consume it, addressing (sector, column, half), not a single, fixed shape. An assembly of several, genuinely different physical boards is supported the same way, via a separate `[multiboard]` file naming each individual board's own config in turn.

Verified, continuing directly from Stage 1/2 above, with no explicit config file at all – `pprender` falls back to a real, small, single-sector default (see `resolve_config_path()` in `breadboard_config(5)` for the full lookup order this follows), and this circuit already fits within it:

```
$ pprender TestSchemes/PoorManAltimeter/altimeter.kicad_sch \
-o altimeter-breadboard.svg
connectivity check: 0 failures
board-bounds check: 0 failures
wrote altimeter-breadboard.svg, 755x413.0
hole-collision check: 0 failures
verify: 0 label overlaps (of 3 labels, checked against all 96
registered regions)
```

A genuinely larger circuit that does not fit the default has two real options: an explicit `--config` file with more room (see `breadboard_config(5)`), or `pprender`'s own `--use-computed` flag, which escalates `auto_layout.py`'s own `compute_optimal_config()` automatically, trying progressively more sectors of the same real shape until the circuit fits or a configurable ceiling is reached.

6 Stage 3b: a new, algorithmically laid-out KiCad schematic

`kicad_layout.py`'s own `generate_schematic()` (or its self-tuning wrapper, `generate_schematic_auto()`, which retries with a more generous sheet size automatically rather than requiring a human to guess one) places anchors in a row, attaches passives to whatever they share a net with, positions PWR_FLAGS clear of real label reach, and verifies the whole result against sheet bounds, the title block, and component-to-component overlap – using `kicad_gen.py`'s own small language underneath (see KicadGen.org for that language's own full reference). The result is written out as a real, valid `.kicad_sch` file: the SAME kind of file Stage 1 started from – closing the loop this document's own title refers to.

CORRECTED 2026-09-16, Péter's own, direct, repeated correction: this stage, as originally written here, had no real command-line tool at all behind it – `generate_schematic_auto()` is a real Python function, called directly, with no `pp*` wrapper of its own. Confirmed directly, not assumed: none of this project's own real command-line tools reach this exact function on this exact path (an existing schematic, directly, to a new one, with no build instructions in between at all) – `ppresolve_components(1)` calls the SAME underlying `generate_schematic_auto()`, but only ever starting from build instructions, never a schematic directly.

CLOSED 2026-10-05: `ppkicad_layout(1)` is that tool – an existing `.kicad_sch`, or a netlist already exported from one (`.net`, the file `ppnetlist(1)` reads), straight to a NEW, algorithmically laid-out `.kicad_sch`, no build instructions in between. Split across a root sheet and child sheets (`OUTPUT-sheetN.kicad_sch`) when above `--max-components`, the per-sheet limit; every file written is reported.

```
$ ppkicad_layout -o tidy.kicad_sch design.kicad_sch
$ ppkicad_layout --max-components 40 \
  --symbol-lib mylib=libs/mylib.kicad_sym \
  -o split.kicad_sch design.net
```

A hierarchical source stays hierarchical (2026-10-05, Péter's own decision): a design whose own sheets the netlist carries (`rp2040-basic-m1: /, /usb-power/, /flash/`) is written as one child sheet per source sheet, named after it, nested as the source is nested – however small, and without being asked. `--flatten` opts out. A source sheet above `--max-components` is cut mechanically inside itself. A flat source is unaffected.

A sheet the source uses more than once stays ONE sheet (2026-10-05, Péter's own decision, option A: source-declared reuse only): when the netlist lists one source file for several sheet paths, the output has one shared child file and several sheet symbols, each instance keeping its own references; instances are matched by their timestamps and the match verified, never guessed. Only for a leaf sheet directly under the top level; anything else is written as independent copies.

A part with several units is laid out unit by unit (2026-10-06): a dual op-amp, a quad gate. A symbol's units are all drawn at the same origin, so as ONE component their

pins sit on top of each other; `ppkicad_layout` makes one virtual component per used unit (`resolve_components.expand_multi_unit()`, key `U3@u2`), each placed as (`unit 2`) under the real reference `U3`. A pin drawn in several units (the LM358N's power pins) is connected in each. The instructions route (`ppresolve_components`) does the same (0.72.28), expanding at the last moment from the real netlist.

The loop on a design that needs a bigger board, with several ground pins (2026-10-07, the knight-roller loop): the instructions state a non-default board (`Board: 7 rows per half ...`) and the parser reads it; a `pin_map` value may be a LIST of real pins; and the breadboard flow wires only POWER nets (rail-like name, or a power pin) to the rails, not every net of more than four members. Open: polarity (FutureResearch.org).

The instructions route can read a design's embedded symbols (2026-10-07): `ppextract_sources SCHEMATIC` writes the symbols the schematic itself embeds, one library file per nickname, and a sources file whose `sym_path` entries point at them (else at an installed library), with an empty `pin_map` for the user – only they know which pin is which. A design that predates a library rename (camera-trap's `Conn_01x03_Male`) can only be resolved this way.

Symbols are looked for in this order: an explicit `--symbol-lib [NICKNAME=]FILE`; then, for a `.kicad_sch` input, the schematic's OWN embedded symbols (and its sub-sheets', Péter's own decision, 2026-10-05: what its author drew with, no library installed anywhere needed); then KiCad's standard libraries. A `.net` carries no symbols, so a project-LOCAL library needs `--symbol-lib` there; a library that cannot be found stops the run up front, naming it and the components that use it. There is no `--sources` file here, unlike Stage 4's `ppresolve_components(1)`: that file exists to supply what a build-instruction text can never carry.

The longer way to the SAME destination – generate build instructions from the existing schematic first (`ppbuild_instructions(1)`), then resolve those into a new one (`ppresolve_components(1)`, Stage 4, below, with a real `--sources` file) – remains, and is the one that also reaches the breadboard.

7 Stage 4: from build instructions back to a breadboard or a new KiCad schematic

Stages 1-3 above all start from a real `.kicad_sch` file. This stage covers the OTHER real starting point this project supports: a human-readable set of build instructions (`ppbuild_instructions(1)`'s own real output) – read back and turned either into a reconstructed breadboard placement, or a genuinely new KiCad schematic, reaching the SAME real destinations Stage 3a/3b do, from the opposite direction.

build instructions (.txt/.org/.md/.html)

- |
- | `ppbuild_instructions(1)` produces this from a real `.kicad_sch`
- | (see Stage 3a)

```

v
ppparse_instructions(1) / ppresolve_components(1)
|
+-----+-----+
|          |          |
| BREADBOARD-ONLY | A NEW KICAD SCHEMATIC
| (verify/inspect) | (needs a real --sources file)
v                v
ppparse_instructions(1)    ppresolve_components(1)
-- prints the reconstructed |
netlist/placement          v
                           a NEW, real .kicad_sch --
                           feeds back into Stage 1/3b, THE LOOP

```

Generating instructions, from this circuit's own real schematic, continuing directly from Stage 1-3's own default board (no `--config` needed) – `ppbuild_instructions(1)`, using its own real, default (plain) format – CORRECTED 2026-09-16 (Péter's own, direct observation, running this exact real command): earlier, this format's own real title header could never be read back by the reverse tools below at all; this is now fixed directly.

```

$ ppbuild_instructions TestSchemes/PoorManAltimeter/altimeter.kicad_sch \
  -o instructions.txt
generated 15 steps (position order, plain format)
wrote instructions.txt

```

1. Place the Seeeduino XIAO (XIAO-SAMD21-SMD, ref U1) straddling the trench, with its leftmost pin in hole a5. It also connects to the power (+) and ground (-) rails.
2. Place the Micro SD Card Module (TRANSFLASH, ref U3) straddling the trench, with its leftmost pin in hole a19. It also connects to the power (+) and ground (-) rails.
3. Place the BMP280_KK (BME280, ref U2) straddling the trench, with its leftmost pin in hole a31. It also connects to the power (+) and ground (-) rails.
4. Connect a jumper wire from hole h11 to hole h22 (net GND).
- ...
14. Connect the LiPo 3.7V 250mAh (BT1) directly to the north rail.
15. Connect a jumper wire from the north rail to the south rail.

Steps 14-15, BT1's own real placement, are explained below.

Reading instructions back, to inspect the reconstructed breadboard placement alone (no KiCad involved at all) – `ppparse_instructions(1)`'s own `--debug` flag additionally annotates each real component's own column/rail membership:

```
$ ppparse_instructions instructions.txt --debug
4 components, 8 nets
BT1: LiPo 3.7V 250mAh
U1: Seeeduino XIAO (anchor, col 5), rail net(s) GND, VBUS
U2: BMP280_KK (anchor, col 31), rail net(s) GND, VBUS
U3: Micro SD Card Module (anchor, col 19), rail net(s) GND, VBUS
...
```

BT1 carries no real (anchor, col ...)~/~(passive, col ...) annotation at all here – it is on neither the main anchor grid nor the ordinary passive one; its own real GND/VBUS net membership is confirmed further down this same real output instead (both real nets list it directly).

A genuine, simple power source (a passive `power_sources.py`'s own `classify_power_sources()` also identifies as a source – a battery being the direct, real example) is placed differently on the breadboard side whenever the board's own configuration has a real rail at all – ADDED 2026-09-16, Péter's own, direct, real-world observation: a real breadboarder never leaves a battery "in plain on the board... very close to a major and complex component" the way an ordinary passive still can, even under this project's own existing rail-attachment fallback (which still reserves a real, if unshared, column on the main grid). Instead: straight to a vertical (left/right) rail when the board has one, distributing from there by jumper to BOTH horizontal rails; otherwise straight to the north horizontal rail, with a single jumper to the south one – exactly steps 14-15, above. This placement, and `ppparse_instructions(1)`'s own reading of it back into a reconstructed breadboard netlist, are both fully supported.

Resolving instructions into a new KiCad schematic IS supported for a rail-direct power source too – CLOSED 2026-09-23, see FutureResearch.org's own now-removed item (ResearchJournal.org's own 2026-09-23 entry has the full, real account) – `ppresolve_components(1)`'s own former, early refusal is gone. Continuing this same, real walkthrough with the altimeter's own instructions regenerated against a board WITH a left rail (`altimeter-with-rail.conf`, `[rail.left] present = true` – see `breadboard_config(5)` for the full format):

```
$ ppbuild_instructions TestSchemes/PoorManAltimeter/altimeter.kicad_sch \
-o instructions-with-rail.txt \
--config TestSchemes/PoorManAltimeter/altimeter-with-rail.conf
generated 16 steps (position order, plain format)
```

Resolving this into a new KiCad schematic needs one more real, direct input the breadboard-only path does not: a `--sources` JSON file, mapping each real ref to where its own real KiCad/Eagle symbol actually lives, plus a `pin_map` entry for every real net name the instructions state that isn't already a bare pin number (an anchor's own SDA/SCL/CS-style net names, and – CORRECTED 2026-09-16, closing a real, direct gap this same session's own rail-membership work opened – "GND" "VBUS" too, for any anchor that now states its own rail membership in prose at all; see `resolve_components(5)`'s own full field reference). A rail-direct power source itself needs NO `pin_map` entry at all for the common

case – `resolve_pin_identities()` (`resolve_components.py`) reads its own resolved symbol's own real pin names directly (BT1's own real `Device:Battery_Cell`, pins "1"+"", "2" "-") and resolves its own real polarity automatically; a `pin_map` entry is only ever needed as a real, explicit fallback, for a symbol whose own real pin names don't confidently match this project's own +- vocabulary at all (`resolve_components(5)`'s own full field reference covers this too):

```
$ ppresolve_components instructions-with-rail.txt \
  --sources TestSchemes/PoorManAltimeter/altimeter-sources.json \
  -o new-altimeter.kicad_sch \
  --anchor-spacing 100 --title "Altimeter workflow demo"
wrote new-altimeter.kicad_sch
wrote new-altimeter.kicad_pro
$ kicad-cli sch erc --severity-error --exit-code-violations \
  new-altimeter.kicad_sch
Found 0 violations
```

Verified, actually closing this loop too: running `ppbuild_instructions` again on that freshly-generated schematic reproduces the identical instruction text (only the title line differs, since it names the source file) – confirming, not merely asserting, that the new schematic really is electrically equivalent to what the instructions described, not merely "some valid-looking schematic".

Not a language reference (KicadGen.org is that, for `kicad_gen.py`'s own placement primitives specifically) and not a rationale for WHY each algorithm works the way it does (ResearchJournal.org's own dated entries are that, including every real bug found building each stage above). This document exists to answer one question directly: given a real schematic, a real netlist, or a real set of build instructions, which real, actual command, in which order, gets you from one real representation to another – confirmed by running it, not assumed.