KicadGen: a small language for scripting KiCad schematics

v0.72.103.244

Péter Felecan

assisted by Claude Sonnet 5.5 (Anthropic)

October 18, 2026

Contents

1 Purpose of this document **2**

2 The role this language plays in the larger toolchain **3**

3 The five-minute mental model **3**

4 Setting up a new schematic **4**
4.1 Versioning the schematic 5

5 Using a component: symbols and pins **5**
5.1 Finding a symbol and its pins 5
5.2 Registering the symbol and its pins 6
5.3 Placing an instance 6

6 Connecting things: wires and labels **7**
6.1 A direct wire, for two things sitting close together 7
6.2 A label, for a net with many members or far-apart connections 7
6.3 Marking a pin as intentionally unused 8
6.4 Annotation text 8

7 The one direction that doesn't fully work **8**

8 Placing things pin-adjacent **8**

9 Routing around something in the way **9**

10 Fitting the page **10**

11 Two things worth knowing about your own environment **10**
11.1 More than one KiCad version might be installed 10
11.2 `kicad-cli` might not be on your `PATH` 10

12 Checking your own work **11**

13 What this language does not do **11**

List of Listings

1 Setting up a new schematic 4
2 Writing the schematic at the end of the script 4
3 Finding the library a symbol is in 5
4 Extracting a symbol's pin offsets with the library's own function 5
5 Registering a symbol and its pins in the PINS dictionary 6
6 Placing an instance 6
7 A direct wire between two pins 7
8 A label for a net with many members 7
9 Marking a pin as intentionally unused 8
10 Annotation text 8
11 A connection routed as an L-shaped path of two wires 9
12 Pointing the example at the right symbol libraries 10
13 Telling make which kicad-cli to run 10

1 Purpose of this document

`kicad_gen.py` is a small Python library — really a tiny domain- specific language — for describing a KiCad schematic in code: place a component, connect two pins, write the file. Any real circuit written using this language (this project's own two example circuits, `ppexample_circuit1~/~ppexample_circuit2`, or a downstream consumer's own real, physically-designed circuit) is written using exactly what's documented below.

This document is a **practical reference** for the language itself, written so that a change to a circuit — add a part, rewire a connection, move something — can be made directly. It does not repeat the **history** of how each rule below was discovered; that narrative, including every bug found along the way and exactly how it was confirmed, lives in ResearchJournal.pdf's own entries and in `kicad_gen.py`'s own module docstring. Read this document to use the language; read those if you want to know why a rule exists.

EXCISED 2026-08-17 from dudhya (old-smartphone revival), the project `kicad_gen.py` was originally developed inside, along with the rest of this project's own generic engine (a separate extraction, 2026-08-14 – see ResearchJournal.pdf's own entries for both dates).

What was genuinely dudhya-specific in the original version of this document (a concrete description of THAT project's own real pipeline, its own real file paths and Makefile targets) was updated and dispatched to dudhya's own DevelopmentStandard.pdf rather than carried forward stale into this project; what remains here is the language reference itself, generalized where an example had named a specific downstream project's own file.

Everything here was true as of v0.72.103.244. Treat `kicad_gen.py`'s own docstring as the final authority if the two ever disagree — code and comment can drift, and the code is what actually runs.

2 The role this language plays in the larger toolchain

`kicad_gen.py` is only ONE piece of this project's own larger toolchain – worth understanding the whole shape before diving into the language itself, since it explains WHY things are organized the way they are. See Concept.org's own "Goal" section for the full picture; briefly:

1. A real netlist (loaded from an existing KiCad schematic via `netlist.py`'s own `load_from_schematic()`, or built directly for a circuit that doesn't have a schematic yet, as this project's own two example circuits do) is the one, shared ground truth.

2. `kicad_layout.py` consumes that netlist to produce a placed, wired, KiCad-valid schematic – entirely algorithmically, using THIS document's own language (`Schematic~/~place()~/~l~wire()`) under the hood, but with every component's own position decided from the netlist rather than typed by hand.

3. A hand-written script (this document's own real subject) can ALSO consume that same language directly, choosing every coordinate deliberately – this is how this project's own two example circuits are built, and how any downstream consumer's own real, physically-designed circuit would typically be authored.

Learning this language well enough to hand-write or hand-edit a circuit script is exactly the skill needed to read or extend `kicad_layout.py`'s own generator code too, since both ultimately call the same `place()~/~label()~/~wire()` primitives – one calling them with hand-chosen coordinates, the other with computed ones.

3 The five-minute mental model

A KiCad schematic is, underneath, a text file (S-expressions, similar in spirit to Lisp) listing every symbol placed on the sheet, every wire drawn between two points, and every text label. Hand-writing that text file is exactly as tedious and error-prone as it sounds — this project tried it and found several real bugs the hard way (see ResearchJournal.pdf). `kicad_gen.py` exists so you never write that text directly: you call Python functions, and the library emits correct, verified S-expressions on your behalf.

Three ideas carry almost the whole language:

1. **Every component is a symbol you "place"** at an (x, y) position on the sheet, in millimetres, optionally rotated. Placing it gives you back a small dictionary mapping each of its pin names to that pin's own absolute (x, y) position — already correctly computed, accounting for the component's rotation.

2. **Two things on the same electrical net are connected either by a real, drawn wire, or by two matching net labels.** A drawn wire is for two things sitting close together, physically. A label is for a net with many members spread across the sheet (like ground), or for two things too far apart to sensibly draw a direct line between.

3. **A label is never placed exactly on a pin.** It sits at the far end of a short stub wire leading away from the pin, so the schematic stays readable rather than looking like every pin has text glued onto it.

4 Setting up a new schematic

Listing 1 is how every script starts: the imports, then the schematic.

Listing 1: Setting up a new schematic

```
import sys, os
sys.path.insert(0, os.path.dirname(__file__))   # so "import kicad_gen" finds
                                                  # the library, from any directory
                                                  # you happen to run the script from
from kicad_gen import Schematic, write_project_file

sch = Schematic(
    title="My circuit",
    date="2026-08-17",
    rev="1.0",             # see "Versioning the schematic" below
    company="my project",
    comments=["One line per title-block comment slot (up to 4)."],
    paper="A4",             # "A4" or "A3"; see "Fitting the page" below
)
```

`sch` is the one object every other call in this language works through. At the end of the script (see Listing 2):

Listing 2: Writing the schematic at the end of the script

```
root_uuid = sch.write("my-circuit.kicad_sch")
write_project_file("my-circuit.kicad_pro", root_uuid, "my-circuit.kicad_pro")
```

`write_project_file` makes the schematic open as a proper KiCad project rather than a bare file — always call it alongside `sch.write()`, passing the `root_uuid` that `write()`

itself returns (the project file needs to reference the schematic's own internal identifier to associate the two correctly).

4.1 Versioning the schematic

Pass the real project version as `rev`, not a hand-typed guess — it will drift out of sync otherwise. This project's own two example circuits (`ppexample_circuit1~/~ppexample_circuit2`) hardcode a fixed `rev` string, since they're synthetic and have no real release cycle of their own; a real, downstream circuit script should instead take it as a command-line argument or read it from a real version file, the same way a project's own build system already tracks its own version, rather than hardcoding a string that will drift.

5 Using a component: symbols and pins

5.1 Finding a symbol and its pins

Every component you place has to already exist in KiCad's own bundled symbol libraries (`/usr/share/kicad/symbols/*.kicad_sym` on a normal install — see "Two things worth knowing about your own environment" below if more than one KiCad version might be involved). You need two things about a symbol before you can use it: which library file it's in, and the (dx, dy) offset of each pin you care about, relative to the symbol's own origin.

To find which library a symbol is in (see Listing 3):

Listing 3: Finding the library a symbol is in

```
grep -l '(symbol "Q_PMOS_GSD"' /usr/share/kicad/symbols/*.kicad_sym
```

(Don't guess the library from the part's **kind** — a generic P-MOSFET symbol turned out to live under `Transistor_FET`, not the `Device` library a similar-sounding resistor or capacitor lives in, and this specific mismatch was found because it differed between KiCad versions; see ResearchJournal.pdf.)

To find a pin's own (dx, dy) offset, open the symbol's block in the library file directly and look for its `(pin ... (at dx dy rotation) ...)` lines, then its `(name "...")` to see which pin that is. For a symbol with very many pins, this is tedious by hand — a short Python snippet using this library's own extraction function is more reliable (see Listing 4):

Listing 4: Extracting a symbol's pin offsets with the library's own function

```
import re
from kicad_gen import extract_and_qualify
block = extract_and_qualify(
    "/usr/share/kicad/symbols/Interface_USB.kicad_sym",
    "STUSB4500QTR", "Interface_USB:STUSB4500QTR")
pin_re = re.compile(
    r'\(pin \w+ line\s*\(at ([\-\d.]+) ([\-\d.]+) (\d+)\)'
```

```
    r'.*?name "(\w+)"', re.DOTALL)
for m in pin_re.finditer(block):
    x, y, rot, name = m.groups()
    print(name, x, y)
```

Important: these (dx, dy) values are in the symbol's own local coordinate system, which uses Y-increases-**upward** — the exact same numbers you read out of the library file are what you put in the `PINS` dictionary below; `place()` does the (non-obvious) conversion to the sheet's own Y-increases-downward convention for you. Do not flip the sign yourself.

5.2 Registering the symbol and its pins

Once you know the library, the symbol name, and its pins, add an entry to a `PINS` dictionary and write a small `place()` wrapper (this project's own two example circuits, via `example_pins.py`, are a real, working illustration of exactly this pattern – extend it rather than duplicating the pattern from scratch) (see Listing 5):

Listing 5: Registering a symbol and its pins in the PINS dictionary

```
PINS = {
    # ... existing entries ...
    "MyNewPart": {"PIN_A": (0, 3.81), "PIN_B": (0, -3.81)},
}

def place(lib_dir_name, bare_name, ref, value, x, y, rot=0):
    full_id = sch.use_symbol(SYM_DIR, lib_dir_name, bare_name)
    return sch.place(full_id, PINS[bare_name], ref, value, x, y, rot)
```

5.3 Placing an instance

Listing 6 places one resistor and gives its reference, value and position:

Listing 6: Placing an instance

```
r1 = place("Device", "R", "R1", "10k", 70, 45)
```

`r1` is now a dictionary: `r1["1"]` and `r1["2"]` are that specific instance's own two pins' absolute sheet coordinates. Every later `wire()` or `label()` call uses these, not the raw offsets from `PINS` — the raw offsets are relative to the symbol's own origin before placement; the dictionary `place()` returns is already in real sheet coordinates.

`rot` (default 0) rotates the whole symbol — and correctly rotates where each of its pins ends up too, worked out for you. Valid values are 0, 90, 180, 270.

6 Connecting things: wires and labels

6.1 A direct wire, for two things sitting close together

Listing 7 joins the reset pin of a chip to the pin of a resistor next to it:

Listing 7: A direct wire between two pins

```
sch.wire(u1["RESET"], r1["2"])
```

Draws one straight line between the two given points. Use this only when the two points are genuinely close and a straight (or simple two-segment) line between them makes sense — see "Placing things pin-adjacent" below for why this matters, and "Routing around something in the way" for what to do when a straight line isn't safe.

6.2 A label, for a net with many members or far-apart connections

Listing 8 puts a label on a pin, and the same name on another pin joins the two:

Listing 8: A label for a net with many members

```
sch.label(r1["1"], "GND", "up")
```

This is the one call that does the most work in this language, so it's worth understanding exactly what happens: a short, fixed- length stub wire is drawn from the pin in the given direction, and a net label reading "GND" is placed at the **far end** of that stub — never on the pin itself. Every other `label()` call anywhere in the schematic using the same net name (here, `"GND"`) is understood by KiCad to be the same electrical net, with no wire needed connecting them across the sheet.

`direction` is one of `"right"`, `"left"`, `"up"`, `"down"` — which way the stub (and the label sitting at its end) points, using this project's own established vocabulary: **up** = toward the top of the sheet (north), **down** = toward the bottom (south), **left*/*right** = toward the west/east edge. Pick the direction that leads **away** from the component's own body, into open space — not back toward it, and not toward a neighbouring pin (see the warning box below).

> **A real trap, found more than once**: if a component has two pins that each get their own `label()` call, their two directions must point **away** from each other, never toward each other. Two stubs pointing at each other converge in the small gap between the pins and cross — which KiCad treats as a genuine electrical junction, silently merging two nets that were never meant to be connected. This exact mistake happened on four different components in one pass alone. If you add a two-pin part with two labelled ends, work out which physical side each pin is actually on (see "Placing things pin-adjacent" below) and point each stub further in that same direction, not back toward its neighbour.

6.3 Marking a pin as intentionally unused

Listing 9 marks a pin as unused on purpose:

Listing 9: Marking a pin as intentionally unused

```
sch.no_connect(u1["SCL"])
```

Use this for a real pin that this specific circuit genuinely does not use — an unused status output, an optional programming pin, and so on — rather than leaving it with no wire and no label at all, which ERC will (correctly) flag as a mistake rather than an intentional choice.

6.4 Annotation text

Listing 10 adds a note to the sheet:

Listing 10: Annotation text

```
sch.text(20, 145, "A note, with\nembedded newlines if you like")
```

Plain text on the sheet, not connected to anything electrically — useful for build notes (”this pin goes to the phone’s own battery connector”, etc.). Embedded newlines are fine and render as real line breaks.

7 The one direction that doesn’t fully work

Confirmed by direct measurement (rendering to SVG and checking where the drawn geometry actually ends up relative to the label’s own anchor point, not just visual impression): `"right"`, `"left"`, and `"down"` all place their label’s text correctly, extending away from the pin as intended. `"up"` does not — no combination of settings tried gave a clean result; the text ends up leaning back toward the pin rather than clearly above it. It still works **electrically** (ERC has no complaint), it just doesn’t look as clean.

Practical guidance: use `"up"` when you must (a pin genuinely on a component’s own north/top edge, with no sensible alternative), but if a different direction is equally natural for a given pin, prefer it. This is an open problem, not a solved one — if you find a fix, it belongs in `kicad_gen.py`’s own `label()` docstring and in DevelopmentStandard.pdf’s Electronics section, not just in your own circuit’s script.

8 Placing things pin-adjacent

A recurring, specific goal across this project’s own schematic reviews: place a component so the pin it connects to is genuinely close by, ideally aligned in at least one axis so the wire between them is a single straight line, not a diagonal one that might cross something else.

The technique: before deciding where to put a new component, place the **other** end of its connection first, and look at that pin's own computed absolute coordinates (the dictionary `place()` already gave you). Then choose your new component's own placement so that, after accounting for its own rotation, its relevant pin lines up in X (for a vertical wire) or Y (for a horizontal one) with the target. Concretely, for a multi-pin IC with several real pins sitting on its own north/top edge: passives wired to those pins were each placed north of the IC, at the same X as the specific pin each one wires to — not south of it, which would have required each wire to travel down and back up, crossing the IC's own body on the way.

Two closely-related things to check afterward, since getting the wire straight doesn't guarantee everything else is fine:

- **Two components can end up nearly on top of each other** if you align both with two pins that are themselves very close together (a real, confirmed case: two capacitors wired to two pins only 2.54mm apart on the same IC ended up only 2.5mm apart themselves). Fix by staggering along the **other** axis — keep the alignment that gives the short, clean wire, but give the two components real separation in the perpendicular direction. See "Checking your own work" below for the specific check that catches this.

- **A straight wire toward a neighbour can pass through unrelated pins** if those pins happen to share the same coordinate on the axis your wire runs along — common on a multi-pin IC, where several pins along one edge share an X or Y value. If this happens, route with two wire segments instead of one straight line (see next section).

9 Routing around something in the way

For a connection that can't safely be a single straight line — because something else sits between the two points — draw it as two (or more) separate `wire()` calls forming an L-shape (or a longer path), moving through an intermediate point that avoids whatever's in the way (see Listing 11):

Listing 11: A connection routed as an L-shaped path of two wires

```
reset_x, reset_y = u1["RESET"]
r1_x, r1_y = r1["2"]
sch.wire((reset_x, reset_y), (r1_x, reset_y))   # sideways first
sch.wire((r1_x, reset_y), (r1_x, r1_y))          # then straight down
```

There is no automatic obstacle avoidance in this language — you choose the intermediate point yourself, based on where you know the obstacle to be.

10 Fitting the page

`paper="A4"` (297mm x 210mm) or `"A3"` (420mm x 297mm) in the `Schematic(...)` constructor. A4 is preferred for anything that reasonably fits — A3 was used briefly for an earlier, much more loosely-spaced layout and found unnecessarily large once spacing was tightened up. There is no automatic scaling or overflow warning — if a component or a label ends up past the page edge, KiCad will still happily export it (onto a second, mostly-empty page, or simply clipped depending on the export), so check this yourself (see "Checking your own work" below).

11 Two things worth knowing about your own environment

11.1 More than one KiCad version might be installed

If `kicad-cli --version` and the KiCad GUI you actually open files in report different version numbers, or if a symbol lookup fails with "substring not found" even though the symbol clearly exists somewhere, this project's own sandbox hit exactly this: an old, apt-installed KiCad 7 provided `/usr/share/kicad/symbols` (the default path), while the actual KiCad 10.0.5 in use for verification was a separately-downloaded AppImage with its own, different bundled symbols — and one symbol (`Q_PMOS_GSD`) had genuinely moved to a different library file between the two versions.

Set the `KICAD_SYM_DIR` environment variable to override `SYM_DIR`'s default and point this tool at a specific symbol source, e.g. a copy extracted directly from an AppImage (`./KiCad.AppImage --appimage-extract`, then look under `squashfs-root/share/kicad/symbols`) (see Listing 12):

Listing 12: Pointing the example at the right symbol libraries

```
KICAD_SYM_DIR=/path/to/correct/symbols python3 ppexample_circuit1
```

11.2 `kicad-cli` might not be on your `PATH`

If you're using a downloaded AppImage rather than an installed package, `KICAD_CLI` (a Makefile variable, separate from the Python-side `KICAD_SYM_DIR` above) needs to name **two words** — the AppImage itself, then `kicad-cli` as the argument telling it which of its bundled tools to run (see Listing 13):

Listing 13: Telling make which kicad-cli to run

```
make KICAD_CLI="/path/to/KiCad.AppImage kicad-cli" kicad-erc
```

12 Checking your own work

Never trust that a script ran without a Python error as evidence the result is correct — none of the checks below are optional, and each one has caught a real, previously-unnoticed bug at least once in this project's own history.

- **Electrical correctness**: `make kicad-erc` (or directly, `kicad-cli sch erc --severity-all -o report.rpt file.kicad_sch`) — read the actual violation count and each specific message, not just whether the command exits successfully.
- **Wire-path collisions**: two unrelated wires geometrically overlapping (not just sharing an intended endpoint) silently merges their nets, and ERC alone does not always make this obvious at a glance. A short script comparing every pair of drawn wire segments for this is worth keeping and re-running after any layout change — see ResearchJournal.pdf, 2026-08-03 entries, for a working example.
- **Label coordinate collisions**: two **differently-named** labels landing on the exact same point is a related, separate check.
- **Component-body overlap**: ERC has no concept of this at all — it is purely visual. Compare every pair of placed components' own coordinates directly; anything closer than roughly 12mm is worth a second look (found a real case at 2.5mm apart this way).
- **Page bounds**: sum up every placed coordinate (symbols, wires, labels, text — but skip the `lib_symbols` block itself, which is full of unrelated internal symbol-definition coordinates that have nothing to do with where things actually ended up on the sheet) and confirm the range fits inside the paper size with a reasonable margin.

None of these are built into this language automatically — they are separate, small scripts written and refined over this project's own history of finding bugs the hard way. Treat them as a checklist to run after any change, not a one-time thing done once and never repeated.

13 What this language does not do

Stated directly, since it would be easy to assume otherwise after reading this far: this is a tool for describing **what connects to what**, plus enough manual control over position and routing to produce something reasonably clean. It does not automatically:

- lay out a whole circuit well on its own when hand-written — every position in a hand-written circuit script is exactly whatever was typed, chosen deliberately by a real, iterative review process, not computed by an algorithm (that's `kicad_layout.py`'s own, separate job – see "The role this language plays in the larger toolchain" above);

- avoid every possible wire crossing or component overlap on its own — the checks above catch problems, they do not prevent them from being possible in the first place;
- check your circuit's own electrical **design** is correct (only that it's wired the way you told it to be) — that's still on you, the datasheet, and ERC's own rule set.

General-purpose, always-correct automatic schematic layout remains, as far as this project's own research has found, an unsolved problem in EDA tooling generally — not a gap specific to this small language. Expect to keep using the checklist above, and expect a genuinely new circuit's first layout attempt to need at least one further round of review.