Library Reference: netlist.py, auto_layout.py, kicad_layout.py, footprint.py, breadboard_lib.py

v0.72.103.244

Péter Felecan

assisted by Claude Sonnet 5.5 (Anthropic)

October 18, 2026

Contents

1 Purpose of this document **2**

2 netlist.py **3**
- 2.1 Component, Net, Netlist . . . 3
- 2.2 Netlist's own query methods . . . 3
- 2.3 Loading a real netlist . . . 4

3 auto_layout.py **4**
- 3.1 Classification, shared by both outputs . . . 4
- 3.2 The Placement class – layout()'s own real output . . . 5
- 3.3 layout(nl, anchor_min_pins=3, rail_threshold=4, anchor_spacing=6, pin_geometry=None, board_config=None, category_footprint=None, forced_anchors=None, max_components=None) -> (Placement, ordered_anchors, passives, rails) . . . 5
- 3.4 Verification – called separately, checked directly, not assumed from a successful return . . . 6
- 3.5 Footprint-aware sizing . . . 6

4 kicad_layout.py **7**
- 4.1 The real PINS contract . . . 7
- 4.2 resolve_pin_key(PINS, bare_name, pin_number, pin_name) -> str . . . 7
- 4.3 label_direction(dx, dy) -> "up" | "down" | "left" | "right" . . . 7
- 4.4 title_block_rect(sheet_w, sheet_h) -> (x0, y0, x1, y1) . . . 7
- 4.5 avoid_label_reach(candidate_x, y, placed_labels, component_centers=(), y_tolerance=6.0, x_margin=3.0, candidate_reach=0.0) -> float . . . 8

4.6 generate_schematic(nl, PINS, sym_dir, ordered_anchors, passives, rails, title, date, rev, company, comments, paper="A4", anchor_y=90, anchor_spacing=48, start_x=45, north_anchor_extra=20, direct_wire_max_dist=50, max_components=None) -> Schematic 8
4.7 generate_schematic_auto(…) -> Schematic 8

5 footprint.py 9
5.1 Pad, Footprint 9
5.2 Footprint's own real methods 9
5.3 Loading a real footprint 10

6 breadboard_lib.py 10
6.1 Core geometry 10
6.2 Net-consistency assertions – a real, direct safety net during rendering itself 10
6.3 Real component shapes 11
6.4 Wires, rails, and labels 11
6.5 Board background and final verification 12
6.6 Real position addressing: sector- and board-aware 12

1 Purpose of this document

A practical reference for this project's own shared libraries' real, public API: which classes and functions exist, what they take, what they return, and the real constraints a caller needs to know. One document, not one per library – these libraries are used TOGETHER, as one pipeline (see Workflow.pdf), not separately; a reader wanting the detailed reference for one stage of that pipeline is very likely already looking at, or about to look at, another.

This is not a repeat of Workflow.pdf (the pipeline, concretely, with real commands and real verified output – read that first if you haven't) and not a repeat of the history each design choice below came from (every real bug found, every real measurement taken – that narrative lives in ResearchJournal.pdf's own dated entries, and this document doesn't re-derive it). Read this document for the API itself; read the other two for why it looks the way it does, and how the pieces actually chain together.

kicad_gen.py – also a library, arguably at a similar level of detail – is deliberately NOT covered here. It already has its own document, KicadGen.pdf, and stays there: it's a genuinely different KIND of reference (a small domain-specific language's own syntax – place a symbol, connect two pins, write a file) from what this document covers (netlist.py's own data model, auto_layout.py's own placement ALGORITHM, kicad_layout.py's own generation orchestration, each with real, non-trivial internal logic a caller needs to understand, not just a function signature to call). Merging the two would leave a document named for one small language covering several genuinely different things under it – kept separate for that reason, not because the two aren't comparably serious pieces of work.

2 netlist.py

The shared, one real ground truth every other module builds on (Concept.org's own "Goal" section) – a plain, structured representation of a KiCad netlist export, independent of whether it came from a real, existing schematic or was built directly in code for a circuit that doesn't have one yet (see Workflow.pdf's own Stage 1 for both real paths, worked).

2.1 Component, Net, Netlist

Three real dataclasses, deliberately plain (no methods beyond a few real query helpers below – everything else in this project reads their own fields directly):

- `Component(ref, value, footprint="", lib="", part="")` – one real component. `lib~/~part` (e.g. `"Device"~/`"C"~) are KiCad's own library/symbol-name pair, needed by the KiCad-side generator to know which real symbol to place; the breadboard side never needs them (it draws generic shapes, not real KiCad symbols). Both default to empty string, so code written before they existed still works unchanged.
- `Net(name, code, nodes=[])` – one real net. `nodes` is a list of `(ref, pin)` pairs, e.g. `("U1", "2")` – exactly how KiCad's own netlist export identifies a pin (by reference designator and pin number/name on that specific symbol), not a synthesized identifier.
- `Netlist(components, nets, pin_names={}, pin_types={})` – the whole circuit. `components` is `ref -> Component`; `nets` is a list of `Net`, in the order KiCad exported them. `pin_names[(ref, pin_number)] -> pin_name` (e.g. `("U1", "2") -> "CC1"`) and `pin_types[(ref, pin_number)] -> KiCad's own electrical pin type string` (e.g. `"power_in"`, `"passive"`) are both additive, defaulted fields – real data KiCad's own export already carries (a `pinfunction~/~pintype` field per node), not inferred or guessed.

2.2 Netlist's own query methods

- `net_for_pin(ref, pin) -> str | None` – which net a given pin belongs to, or `None` for a genuine no-connect.
- `pins_on_net(name) -> list[(ref, pin)]` – every member of a named net, or `[]` if no net by that name exists.
- `refs() -> list[str]` – every component reference designator, sorted.
- `nets_needing_pwr_flag() -> list[str]` – every net KiCad's own ERC would flag as an undriven power input: at least one `power_in` pin, and no `power_out` pin anywhere on the same net. Derived directly from real pin electrical types, not a hand- picked net-name list – confirmed against this project's own real circuit before being relied on (every net the rule predicted matched a hand-verified, real PWR_FLAG list exactly).

2.3 Loading a real netlist

- `load_netlist(text) -> Netlist` – parses `kicad-cli sch export netlist --format kicadsexpr` output (already read into a string) into a `Netlist`. Raises `ValueError` if the top-level form isn't `(export ...)` – a genuine "this isn't a netlist export" signal, not a silent, partial parse.

- `load_from_schematic(kicad_sch_path, kicad_cli="kicad-cli") -> Netlist` – the real, usual entry point: runs `kicad-cli sch export netlist` on a real `.kicad_sch` file and loads the result directly, managing the intermediate file itself. `kicad_cli` may be a multi-word string (e.g. `"./KiCad.AppImage kicad-cli"` for an AppImage install – split on whitespace before being passed to `subprocess`), matching this project's own `$KICAD_CLI` convention (see netlist.py(1), render.py(1)). Hierarchical, multi-sheet schematics are followed automatically by `kicad-cli` itself, provided every real sheet file is still present alongside the root – see ResearchJournal.pdf'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 at all).

3 auto_layout.py

This project's own placement engine, shared by both real outputs (Concept.org's own "Goal" section): `classify()~/~high_degree_nets()~/ ~net_members()~/~order_anchors_by_connectivity` feed BOTH the breadboard side (this module's own `layout()`) and the KiCad side (`kicad_layout.py`'s own `generate_schematic()`); `layout()` itself is breadboard-specific. The most complex file in this project – this section covers its own real, callable public contract, not every internal helper; the internal placement mechanics (BFS attachment, room-checking, the far-leg fallback, declutter/push- aside/relieve_overflow) are extensively commented in the source itself and narrated, bug by bug, in ResearchJournal.pdf.

3.1 Classification, shared by both outputs

- `classify(nl, anchor_min_pins=3) -> (anchors, passives)` – every component with `anchor_min_pins` or more real net memberships is an anchor; everything else is a passive.

- `high_degree_nets(nl, threshold=4) -> set[str]` – real net names with more than `threshold` members (GND/power rails in a typical circuit) – excluded from placement-driving connectivity, wired via a rail jump instead (see this section's own "rails" note below, and FutureResearch.pdf's own rail-attachment item for a real, known limit of this member-count heuristic).

- `net_members(nl, ref, exclude_nets) -> [(net_name, pin)]` – every real net `ref` belongs to, skipping `exclude_nets` (usually the rail set) and unconnected pseudo-nets.

- `other_net_of(nl, ref, attached_net) -> str | None` – a passive's own OTHER real net (not the one it attached through). Correct only for a genuinely 2-real-net component – true of every passive by construction, since `classify()`'s own threshold means anything with 3+ real nets is an anchor before this is ever called on it.
- `pin_sort_key(pin) -> (int, str)` – numeric pins sort numerically, non-numeric ones after, alphabetically.
- `pin_counts(nl) -> {ref: int}` – real net-membership count per component (what `classify()`'s own threshold is compared against).
- `order_anchors_by_connectivity(nl, anchors, rails) -> [ref]` – anchors ordered so electrically-related ones land physically close together, not reference-designator order – confirmed a real source of wire clutter once every connection was actually drawn: alphabetically adjacent anchors aren't necessarily electrically related.

3.2 The Placement class – layout()'s own real output

Deliberately explicit, not opaque, so the result can be inspected and criticized, not just rendered and eyeballed (render.py, ppauto_layout(1), and ppvisualize(1) all read these fields directly): `anchor_col~/~anchor_pin_col~/~anchor_pin_half` (an anchor's own starting column and each of its own real pins' exact column/half), `passive_col~/~passive_half~/~col_row` (which column/half/row a passive landed at), `size_rows` (its own real, footprint-derived leg span, computed during placement itself so room-reservation uses the real figure, not a guess), `reason` (a real, human-readable trace of WHY – which net, attached to which other component, including any relocation), `unplaced` (components the algorithm genuinely could not place – an honest report, not a silent drop), `anchor_rail_col~/ ~anchor_rail_half` (an anchor's own real rail-jump connections), `extra_links` (explicit bridging wires where column-sharing alone didn't unify every real member of a net).

3.3 layout(nl, anchor_min_pins=3, rail_threshold=4, anchor_spacing=6, pin_geometry=None, board_config=None, category_footprint=None, forced_anchors=None, max_components=None) -> (Placement, ordered_anchors, passives, rails)

The real breadboard placement algorithm: anchors placed in a row (ordered by connectivity), each of their own real pins getting its own column (no wrapping – confirmed a real bug source when it used to estimate a fixed width instead); passives attached via BFS to whatever already-placed member (anchor or passive) they share a real net with, including a passive's own far pin, not just its near one; room-checked throughout via `_find_room()` (searches nearby columns for real space before committing, relocating rather than silently overflowing); declutter/push-aside/relieve_overflow resolve visual crowding and any remaining overflow afterward. `pin_geometry`, when given a real `{(ref, pin):`

(dx, dy)} table, lets north/south splitting use a real anchor's own genuine top/bottom pin grouping instead of pure list position – falls back to the positional split for any pin missing from the table, so the algorithm stays usable with no real geometry at all. board_config, when given a real BoardConfig or MultiboardConfig, assigns each anchor's own connectivity group to a real sector (offline, first-fit-decreasing bin-packing: groups sorted by descending width, each placed into the first sector with enough remaining room, opening a fresh sector when none fits) – defaulting to a single, plain sector's own geometry otherwise. max_components, when given, overrides this project's own real, default component-count ceiling (breadboard_config. DEFAULT_MAX_COMPONENTS, 150) – raises TooManyComponentsError immediately, before any real placement work at all, whenever nl's own real, total component count exceeds whichever limit is in effect (see that constant's own docstring for the real, direct rationale). generate_schematic()~/~generate_schematic_auto() (kicad_layout.py) carry the SAME real check, reusing the SAME real constant, since layout() is never actually called on that, separate, KiCad-schematic generation path at all.

Rails: a component touching a high-degree net attaches via rail_nets_touched() (which real rails a ref touches) and a direct rail jump, not column-sharing – tried for both anchors and passives, so a passive whose entire connectivity routes through rail nets still gets a real attach path.

3.4 Verification – called separately, checked directly, not assumed from a successful return

- verify_within_board_bounds(p) -> (bool, [str]) – every assigned row, and every extra_links wire's own landing row, checked against the real, parametric ROWS_PER_HALF limit.
- verify_connectivity(nl, p, rails) -> (bool, [str]) – every real net's own members confirmed to form one genuinely connected group in the actual layout (shared column, or bridged via extra_links), not merely "each individually attached to something."

3.5 Footprint-aware sizing

- guess_shape(value) -> str – a real component category ("resistor", "capacitor", …) guessed from its own Value string, when no real footprint data is available.
- real_size_rows(value, fallback_rows, ref=None, assume_resistor_default=True) -> int – prefers a real footprint's own courtyard-derived row span (via footprint.py's load_footprint()) over a guessed one; fallback_rows (usually guess_size_rows(value)) is used only when no real footprint data resolves.
- guess_size_rows(value) -> int – the fallback heuristic itself, used when $KICAD_FOOTPRINT_DIR isn't set or a specific footprint can't be resolved.

4 kicad_layout.py

Consumes a real `Netlist` (already classified via `auto_layout.py`'s own `classify()~/~high_degree_nets()` circuit-agnostic and unrelated to this module) and produces a fully placed, labeled, VERIFIED KiCad `Schematic` – entirely algorithmically, using `kicad_gen.py`'s own language underneath (see KicadGen.pdf for that language itself). This project's own PRIMARY KiCad generator (Concept.org's own "Goal" section).

4.1 The real PINS contract

Every function below that places a real anchor symbol needs a `PINS` dict: `PINS[part_name][pin_key] -> (dx, dy)`, a real, hand-verified millimeter offset from that symbol's own origin to each of its own real pins – see example_pins.py for this project's own real, confirmed entries, and ResearchJournal.pdf's own 2026-08-18 entry for a real, concrete feasibility check of extracting one for a genuinely new symbol. There is no general, automatic way to populate this dict for an arbitrary symbol built into this module – a real, known limitation, not an oversight (see FutureResearch.pdf's own "remaining scripts" item for why this module has no CLI of its own yet).

4.2 resolve_pin_key(PINS, bare_name, pin_number, pin_name) -> str

Which real key into `PINS[bare_name]` corresponds to a given real pin – tries the plain pin number, then the plain pin name, then name+number concatenated or underscore-joined (how a hand-built PINS dict disambiguates same-named pins on one symbol, e.g. two real GND pins). Raises `KeyError`, listing every form actually tried, if none matches – confirmed necessary, not a defensive guess: a plain-name lookup alone silently failed against a real, disambiguated PINS dict this module was tested against.

4.3 label_direction(dx, dy) -> "up" | "down" | "left" | "right"

Which way a pin's own label should point, derived from that pin's own real offset: whichever AXIS dominates for that specific pin (not dx checked first unconditionally) – confirmed the hard way: checking dx's sign first is wrong whenever a pin sits near a symbol's own top/bottom edge without being perfectly centered, and KiCad's own ERC caught the resulting real short directly (two labels landing at the same point).

4.4 title_block_rect(sheet_w, sheet_h) -> (x0, y0, x1, y1)

The title block's own real rectangle for a sheet of the given size. CONFIRMED empirically: KiCad's own title block is a FIXED-size box (108.0 x 32.0mm, 12.0mm margin), anchored to the sheet's own bottom-right corner, independent of paper size – verified by

generating two otherwise-identical real test schematics at different paper sizes and measuring each one's own title block directly from KiCad's own SVG export.

4.5 avoid_label_reach(candidate_x, y, placed_labels, component_centers=(), y_tolerance=6.0, x_margin=3.0, candidate_reach=0.0) -> float

Nudges a candidate X position (a fixed-row placement, e.g. a PWR_FLAG) clear of any already-placed LEFT/RIGHT label's own real, measured reach, AND clear of any already-placed component's own real Value text – both measured, not guessed (see `kicad_gen.py`'s own `label_reach()~/~value_text_reach()`). `component_centers` was added after a real, direct SVG render found a flag landing squarely on an unrelated component's own Value text – `label_reach()` alone never covered that case.

4.6 generate_schematic(nl, PINS, sym_dir, ordered_anchors, passives, rails, title, date, rev, company, comments, paper="A4", anchor_y=90, anchor_spacing=48, start_x=45, north_anchor_extra=20, direct_wire_max_dist=50, max_components=None) -> Schematic

The full, real placement algorithm: places every anchor in a single row (ordered by shared connectivity), attaches every passive to whatever anchor or other passive it shares a net with, positions PWR_FLAGs clear of real label reach, verifies the whole result (sheet bounds, title block, component-to-component overlap) before returning. Raises `RuntimeError` – not a silent bad file – if any placed label overflows the sheet's own printable area, or any component lands inside the title block. Raises `UnplaceableError` (a real, distinct exception) for a genuine topology problem – a passive whose only real net connects to another never-placed passive, or to nothing – since no amount of extra sheet room can ever fix that.

`anchor_y~/~anchor_spacing~/~start_x~/~north_anchor_extra` are real parameters, not hidden constants, but their own DEFAULTS are tuned to this project's own real, original circuit – see ResearchJournal.pdf for that history, and `generate_schematic_auto()` below for the real, working answer to "what if a different circuit needs different values."

4.7 generate_schematic_auto(...) -> Schematic

Same real contract as `generate_schematic()`, minus `paper~/~anchor_y~/~north_anchor_extra` – chosen automatically instead of requiring the caller to guess. Retries with a genuinely more generous tier – `PAPER_LADDER = ["A4", "A3", "A2", "A1", "A0"]`, and for each paper size, `NORTH_EXTRA_LADDER = [20, 40]` – only when the placement genuinely needs more room, not on every call regardless. `anchor_y` is re-derived from each tier's own real paper

height (keeping the same real proportion this project's own original circuit is tuned to), not held fixed across tiers. `UnplaceableError` is NEVER retried – it propagates immediately, matching its own real meaning above. Raises `RuntimeError` with the FULL real history of every tier tried and why each one failed if even the most generous tier still doesn't fit – a genuine "this circuit needs a human to look at it" signal, not a silent failure. `auto_layout.py`'s own `compute_optimal_config()` follows this same escalating-retry-ladder pattern on the breadboard side: an assembly's own base, single-sector shape stays fixed while the sector count alone escalates, 1 up to a given ceiling, retrying `layout()` at each tier and checking `verify_within_board_bounds()~/ ~verify_connectivity()` directly, returning the first (fewest-sectors) config that genuinely fits, or raising with the full history of every tier tried if none does – the real, physically meaningful parameter for an assembly built from several identical units. `pprender`'s own `--use-computed` flag exposes this directly.

5 footprint.py

Loads a real KiCad footprint (`.kicad_mod`) file and provides its own real pad geometry and courtyard size – the same data `auto_layout.py`'s own `real_size_rows()` uses for footprint-aware component sizing, and `footprint.py(1)`'s own command-line use wraps.

5.1 Pad, Footprint

- `Pad(number, shape, kind, x, y, size_x, size_y)` – one real pad. `shape` is KiCad's own string (`"circle"`, `"oval"`, `"rect"`, `"roundrect"`, …); `kind` is `"thru_hole"` or `"smd"`; `x~/~y~/~size_x~/~size_y` are all real millimeters, relative to the footprint's own origin.

- `Footprint(name, description, pads=[], courtyard_bbox=None)` – one real footprint. `courtyard_bbox` is `(x0, y0, x1, y1)` in mm (KiCad's own standardized "real keep-out size" layer), or `None` if the footprint has no courtyard layer at all.

5.2 Footprint's own real methods

- `pad_span_x() -> float` – horizontal distance between the leftmost and rightmost pad CENTERS. For a simple 2-pad part, this IS the pitch; for anything with more pads, it's the overall pad spread. Returns `0.0` for fewer than 2 pads.

- `bounding_box() -> (width, height)` – covers every pad's own real EXTENT (not just its center point) – a truer "how much physical room does this part's own copper need" than `pad_span_x()` alone.

- `courtyard_size() -> (width, height) | None` – derived from `courtyard_bbox`; `None` if that's `None`. Kept as a separate convenience method so a caller wanting only the size, not the exact bounding coordinates, doesn't need to unpack a tuple itself.

5.3 Loading a real footprint

- `find_footprint_file(fp_ref) -> str` – resolves `"LibraryName:FootprintName"` (KiCad's own exact reference form, the same string a real component's own `footprint` field holds) to a real `.kicad_mod` path, searched under `$KICAD_FOOTPRINT_DIR` if set, else `/usr/share/kicad/footprints`. Raises `ValueError` for a malformed reference (no `:`), `FileNotFoundError` if nothing matches.
- `load_footprint(fp_ref) -> Footprint` – resolves via `find_footprint_file()` and parses every real pad and the courtyard layer, if present. Raises `ValueError` if the file isn't a real `(footprint ...)` form.

6 breadboard_lib.py

Real SVG drawing primitives and the geometry constants `breadboard_config.py`'s own `default_config()` mirrors byte-for- byte (`COL_SPACING~/~ROW_SPACING~/~ROWS_PER_HALF~/~TRENCH_GAP` became the FIRST sector's own real config, once multi-sector geometry existed – see this section's own "Multi-sector geometry" note below).

6.1 Core geometry

- `COL_SPACING~/~ROW_SPACING~/~ROWS_PER_HALF~/~TRENCH_GAP~/ ~LEG_ROWS` – the real, module-level pixel constants every shape and position calculation below is built from.
- `col_x(col) -> float`, `row_y(half, i) -> float` – the real pixel position of a given column, or a given row within a half (`"N"` or `"S"`). The two functions every other real position in this module (and every position `render.py` computes) ultimately comes from.
- `row_letter(half, i) -> str` – a real, human-readable row label (matching a real breadboard's own printed row lettering).
- `RAIL_PLUS_TOP~/~RAIL_MINUS_TOP~/~RAIL_PLUS_BOT~/~RAIL_MINUS_BOT` – derived directly from `ROWS_PER_HALF~/~row_y()`: the real Y position of the single board's own top and bottom rail pair.
- `max_col()~/~rightmost_edge()` – how wide the CURRENT drawing has actually grown (tracks real, placed content, not a fixed board width – see LibraryReference.org's own auto_layout.py section for why there's no hard column limit today).

6.2 Net-consistency assertions – a real, direct safety net during rendering itself

- `set_netlist(netlist)` – registers the real Netlist a rendering pass is drawing against, so the assertions below can check a real pin's own net membership, not trust the caller

blindly.

- `assert_same_net(pin_a, pin_b)`, `assert_net_named(pin, expected_name)` – real, direct checks called throughout the shape-drawing functions below, catching a wiring mistake AT THE POINT it would be drawn, not only later via `verify_connectivity()`.

- `report_net_failures(source_description)` – prints every assertion failure collected so far, with real context.

6.3 Real component shapes

`resistor()~/~capacitor()~/~diode()~/~inductor()` (all built on `leg()`, the shared near/far-leg drawing primitive – real and enforced: near and far legs must land in DIFFERENT columns, or a component's own two different nets would short together through the shared column itself), plus `ic()` for a multi-pin symbol. Each takes a real column/row position and returns the real pixel position of its own far leg (or far pins, for `ic()`) – used directly by `render.py`'s own far-leg-jumper pass (see LibraryReference.org's own auto_layout.py section).

6.4 Wires, rails, and labels

- `jumper(x1, y1, x2, y2, color, bulge="N", height=20, avoid=False)` – a real bridging wire between two arbitrary pixel positions, with real collision-avoidance (`avoid_label_reach()`-style nudging) when `avoid` is set.

- `to_rail(x, half, plus, color, col=None, board_config=None)` – a real connection from a column to the assembly's own rail, in order of preference: a direct connection to the nearest shared horizontal rail when one is configured; otherwise, a jumper to the assembly's own left vertical rail when one is configured (a genuine last resort, tracked per real, physical board so a densely-connected board earlier in a multi-board assembly can never push a later board's own connections past where they should start); otherwise, the single, module-level default. The assembly's own right vertical rail has no direct connection tier here yet – a real, current limitation, not an oversight: its own X position genuinely depends on the assembly's own final, dynamically-determined width, not yet known at a `to_rail()` call's own time.

- `bus(col_list, y_list, net_label, color, board_config=None)` – a real high-degree net's own bus line, drawn once rather than as N separate wires.

- `label()~/~part_label()~/~flush_labels()` – real text labels, with real collision-avoidance against everything already registered (`_register()~/~_overlaps()~/~_overlap_depth()~/~_clearance()` – module-private, but architecturally real: every shape above registers its own real occupied region here, which is what lets later placements detect and avoid overlap).

6.5 Board background and final verification

- `draw_board_background(N_COLS, W, H)` – the real board outline, trench, and hole grid itself, drawn once behind everything else, including any configured left/right vertical rail, per real, physical board in a multi-board assembly.
- `verify() -> (bool, [str])` – real hole-collision checking: every real lead's own final position checked against every other for two different leads landing in the same physical hole, plus real label-overlap checking against every registered region.

6.6 Real position addressing: sector- and board-aware

Every real shape function with an internal `col_x()~/~row_y()~/ ~TRENCH_Y` dependency – `ic()~/~to_rail()~/~bus()~/ ~draw_board_background()` – accepts an optional `breadboard_config.BoardConfig` (or `MultiboardConfig`, for an assembly of several, genuinely different physical boards) and correctly offsets by sector and by board, defaulting to a single, plain sector's own geometry otherwise. `render.py`'s own `_x()~/ ~_y()` wrappers do the same conversion for its own direct `col_x()~/~row_y()` calls.