

LUMICRON

PHOTONIC LAYOUT, FROM PYTHON.

User Guide

The lumicron Python package

Edition	October 2026
Documentation	6 October 2026
Publisher	Elyon Semiconductor

Lumicron User Guide

© 2026 Elyon Semiconductor. All rights reserved.

Documentation for the Python API bundled with Lumicron.

Chapters, examples, figures and API signatures are preserved from the current application guide sources.

Web edition: <https://piclumicron.com/docs/>.

Use the documentation matching your installed release.

Table of contents

Welcome	1
Package ownership	1
Preface	3
Who this is for	3
What's covered	3
What's <i>not</i> covered	4
Conventions	4
How to read	4
I. Getting Started	5
1. Getting started	6
1.1. Run in Lumicron	6
1.2. Hello, Lumicron	6
1.3. A more interesting first chip	8
1.4. What you just used	10
1.5. Phoebe Guide and native circuit experiments	10
1.6. Code → Schematic	12
2. Core concepts	14
2.1. Naming convention	14
2.2. Layers	15
2.3. Cells	16
2.4. Ports	17
2.5. Inspection and physical export	17
2.6. Cell information	17
2.7. Package ownership	18
II. Building Layouts	19
3. Placement	20
3.1. The chain	20
3.2. Anchors	22
3.3. Joining ports	23

3.4. Arrays	24
3.5. Hierarchy and re-use	25
3.6. Promoting child ports	26
3.7. Positioning order and returns	26
3.8. Transform order	27
4. Routing	28
4.1. Auto-routing	28
4.2. Bends	31
4.3. Manual routing with <code>.go()</code> and <code>.jog()</code>	31
4.4. Routing bundles	33
4.5. Length matching	36
4.6. Section-level overrides	39
4.7. Tapers	41
4.8. Layout-aware routing	41
4.9. Endpoint straights and loopbacks	42
4.10. Endpoint requests and compatibility	42
4.11. PDK width permission and transitions	43
4.12. Inspecting and completing routes	43
5. Route profiles	45
5.1. Using a profile	45
5.2. What a profile contains	47
5.3. When to override at the call site	47
5.4. Auto layer transitions	47
5.5. Where profiles come from	49
6. Bookmarks	50
6.1. Two ways to declare a bookmark	50
6.2. Groups	51
6.3. Notes	51
6.4. Auto-named bookmarks on placements	52
6.5. Empty names are rejected	52
6.6. Notebook Notes	52
7. Registry and physical test manifests	54
7.1. Declare named test ports	54
7.2. Existing physical I/O sites	55
7.3. Identity, hierarchy and validity	56
7.4. Native Registry Viewer	56
7.5. Deterministic exports	57
7.6. Persistence and freshness	57
7.7. Complete qualification example	58
III. Working with PDKs	59
8. Working with PDKs	60
8.1. Importing	60

8.2. Discovering what's in a PDK	61
8.3. Using PDK components	61
8.4. Component ports	61
8.5. Loading a PDK in Lumicron	62
8.6. Writing your own PDK	62
9. Packages and project environments	63
9.1. Inspect an environment in Lumicron	63
9.2. Start a project with local and Git dependencies	64
9.3. A tiny process and its component library	65
9.4. A tapeout consumes packages	66
9.5. Requirements, lock, and local edits	67
9.6. Git sources and offline reconstruction	68
9.7. Multiple processes, components and validation	68
9.8. Organize your repository freely	69
9.9. Identity and generated evidence	69
9.10. Project branches, conflicts, and recovery	70
9.11. Run the project pipeline locally	70
IV. Cookbook	72
10. Cookbook	73
10.1. Mach-Zehnder interferometer	73
10.2. Ring resonator	75
10.3. Fanout bus	76
V. Agent Workflows	78
11. Working with Phoebe	79
11.1. Reference questions	79
11.2. Project questions	79
11.3. Actions and reasoning	79
Appendices	81
A. Cheat sheet	81
A.1. Boilerplate	81
A.2. Cells & shapes	81
A.3. Placement	82
A.4. Ports	82
A.5. Routing - global kwargs	83
A.6. Routing - chain methods	83
A.7. Routing bundles	83
A.8. Bookmarks	84
A.9. Output	84
A.10. Useful introspection	84

B. Errors & warnings	85
B.1. Errors	85
B.2. Warnings	86
B.3. Suppressing warnings	87
C. Glossary	89
API reference	91

Welcome

This is the User Guide for **lumicron** - the Python package that lets you describe a photonic or electronic chip layout in code and export a GDS or OASIS layout. Export alone does not establish foundry-rule compliance or fabrication readiness.

If you've used IPKISS, gdsfactory, KFactory, or Nazca, the mental model will feel familiar:

- **Cells** are reusable building blocks with named **ports**.
- **add** creates placed references; **Place** positions them; **Route** connects ports.
- **Route profiles** describe what travels along the wire (layers, claddings, periodic features) so you stop thinking about per-stroke geometry.

What's different in Lumicron is the routing chain - section-level overrides for layer, profile, bend, and width inside one continuous `.go()` chain - and the design-tool integration with the Lumicron Mac viewer. Scripts run in the Script workspace; Phoebe can inspect permitted project context and retrieve the bundled API reference.

This guide covers the Python API shipped with Lumicron and its scripting workflow. Authoring your own PDK is covered in the separate PDK Authoring Guide.

Tip

The numbered layout examples in `user_docs/examples/` are executed against the bundled API when this guide is built, and their figures are regenerated. Short inline examples demonstrate an API call and may require objects defined in the surrounding text. The API appendix is generated from current signatures.

Package ownership

Use `import lumicron as lm` for PIC design. Technology authoring and packaging belong to `import lumicron_pdk as lpdk`; fabrication rules and semantic DFX belong to `import lumicron_validate as lv`. All three ship together in the bundled runtime. See the [PDK Development guide](#) and [Validation guide](#).

The former package remains a deprecated forwarding bridge for existing projects and persisted factory references. New code uses the canonical imports. Placement, routing, transforms and authored identities retain their established contracts.

Preface

Who this is for

You're a chip designer - photonic, electronic, or both - who wants to write layouts in Python instead of clicking around a layout editor. You know what a GDS file is. You probably know what a port is. You may already know one of the existing layout DSLs and are looking for either a faster authoring experience, a tighter design-tool integration, or both.

You don't need to be a Python expert. The library is shaped to read like English: `c.add(...)`, `c.Place(thing).at(...)`, `c.Route(a, b).go("E", by=200)`. Where the names are unusual, this guide explains why.

What's covered

- **Getting started** (Chapters 1–2): the bundled runtime, the smallest possible script, the core nouns and verbs of the API.
- **Building layouts** (Chapters 3–6): placement, routing (auto and manual), route profiles, Bookmarks, and Notes.
- **Working with PDKs** (Chapter 7): using a process design kit's layers and components.
- **Cookbook** (Chapter 9): full recipes for an MZI, a ring resonator, and a fanout bus.
- **Agent workflows** (Chapter 10): Phoebe reference lookup, project inspection, and action permissions.
- **Packages and environments** (Chapter 8): consuming local/Git engineering packages through explicit project dependencies.
- **Appendices**: a compact cheat sheet, errors and warnings, a glossary, and generated API signatures.

What's *not* covered

- **Writing your own PDK** - that lives in the *PDK Authoring Guide*, which also covers the `@route_profile` builder for custom cross-sections.
- **The Lumicron Mac viewer's GUI** - see the in-app help.
- **Internal architecture** - this guide focuses on the public authoring API.

Conventions

Code blocks appear as:

```
import lumicron as lm
c = lm.CELL("Hello")
```

Three callout styles signal different intents:

Note

A clarification about why something is the way it is, or a side-effect worth knowing.

Tip

A pattern that shortens your script or avoids a common gotcha.

Warning

Something that will trip you up if you're not aware of it.

How to read

Chapters 1 and 2 build vocabulary; everything after is reference plus recipes. Skim the cheat sheet (Appendix A) once before reading the rest - it tells you which idioms to expect.

Part I.

Getting Started

CHAPTER 1

Getting started

1.1 Run in Lumicron

Lumicron includes Python 3.12 and its matching `lumicron` package. Create or open a project, create a Python file in Script, and use Run. No separate Python installation is required for the bundled runtime. For standalone development, use an environment containing this same API and its dependencies; an unrelated installed package may expose a different API.

The examples use the bundled educational `elyon_demo` PDK. It demonstrates the API and is not a fabrication-qualified foundry process. Use your project's installed PDK for real designs.

1.2 Hello, Lumicron

The smallest meaningful script: one cell, one rectangle, one GDS file. The explicit export in these standalone examples also works from the app.

`01_hello_world.py`

```
"""Hello, Lumicron – your first cell.

The smallest meaningful Lumicron script: one cell, one rectangle, one
GDS file. If this runs, your install is healthy.
"""
import lumicron as lm

METAL = lm.LayerSpec(layer=4, datatype=0)

@lm.pcell
def HelloLumicron():
    c = lm.CELL("HelloLumicron")
```

```
pad = lm.Rectangle(x_dim=80, y_dim=80, layer=METAL)
c.add(pad)
c.Place(pad).at((0, 0))
return c

if __name__ == "__main__":
    HelloLumicron().to_gds("hello_lumicron.gds")
```

Run it:

```
python 01_hello_world.py
```

If `01_hello_world.gds` lands in your working directory, you're set up. Open it in any GDS viewer (KLayout, the Lumicron Mac app, your foundry's tool) and you'll see an $80 \times 80 \mu\text{m}$ metal pad at the origin.

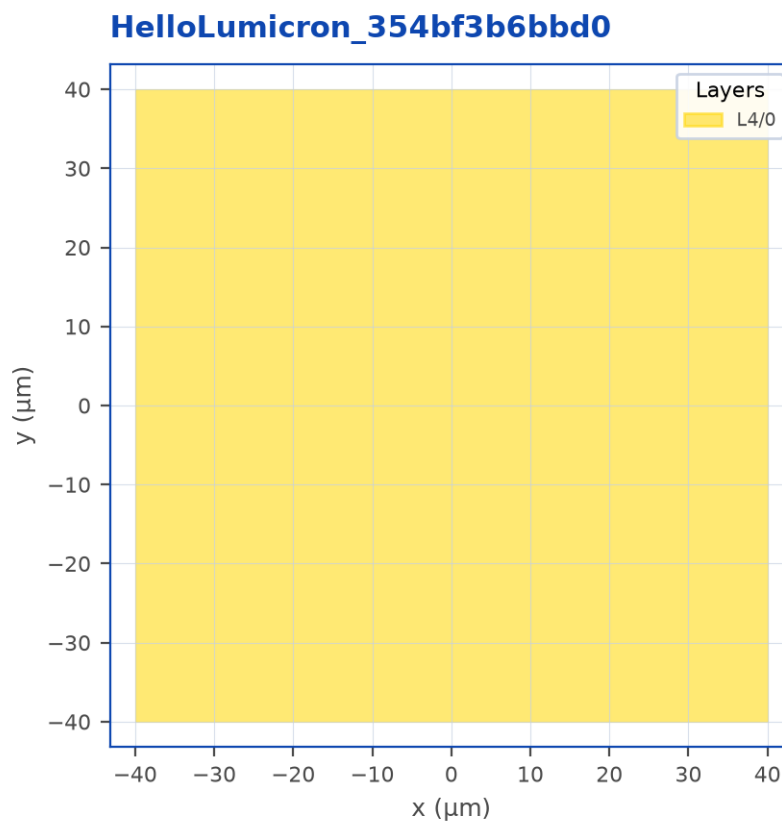


Figure 1.1.: Output of `01_hello_world.py`.

i Note

The file name `hello_lumicron.gds` would have been auto-derived from the cell name if you'd called `chip.to_gds()` with no argument - `to_gds()` lower-snake-cases the cell name when no path is given.

1.3 A more interesting first chip

Real layouts have multiple shapes on multiple layers. This one adds a die outline, a ring resonator, and two metal pads.

02_first_chip.py

```

"""A frame, a ring, two pads – the canonical first chip.

Three concepts in one file:
    1. Multiple shapes on different layers.
    2. Placement at explicit coordinates.
    3. Hierarchy: this cell is reusable later.
"""

import lumicron as lm

OUTLINE = lm.LayerSpec(layer=63, datatype=0)
METAL = lm.LayerSpec(layer=4, datatype=0)
SILICON = lm.LayerSpec(layer=9, datatype=0)

@lm.pcell
def FirstChip():
    c = lm.CELL("FirstChip")

    # Die outline: 2 mm × 1 mm. Place anchored at the south-west corner
    # so the chip occupies (0, 0) → (2000, 1000).
    frame = lm.Rectangle(x_dim=2000, y_dim=1000, layer=OUTLINE)
    c.add(frame)
    c.Place(frame).using("SW").at((0, 0))

    # A silicon ring resonator, off-center.
    ring = lm.Ring(outer_radius=50, width=4, layer=SILICON)
    c.add(ring)
    c.Place(ring).at((600, 500))

    # Two metal pads. We capture each handle from c.add() so each call
    # to Place() targets a distinct instance – re-using the same `pad`
    # spec without capturing would just move the most-recent placement.
    pad = lm.Rectangle(x_dim=80, y_dim=80, layer=METAL)
    pad_top = c.add(pad)

```

```

pad_bot = c.add(pad)
c.Place(pad_top).at((1700, 750))
c.Place(pad_bot).at((1700, 250))
return c

if __name__ == "__main__":
    FirstChip().to_gds("first_chip.gds")

```

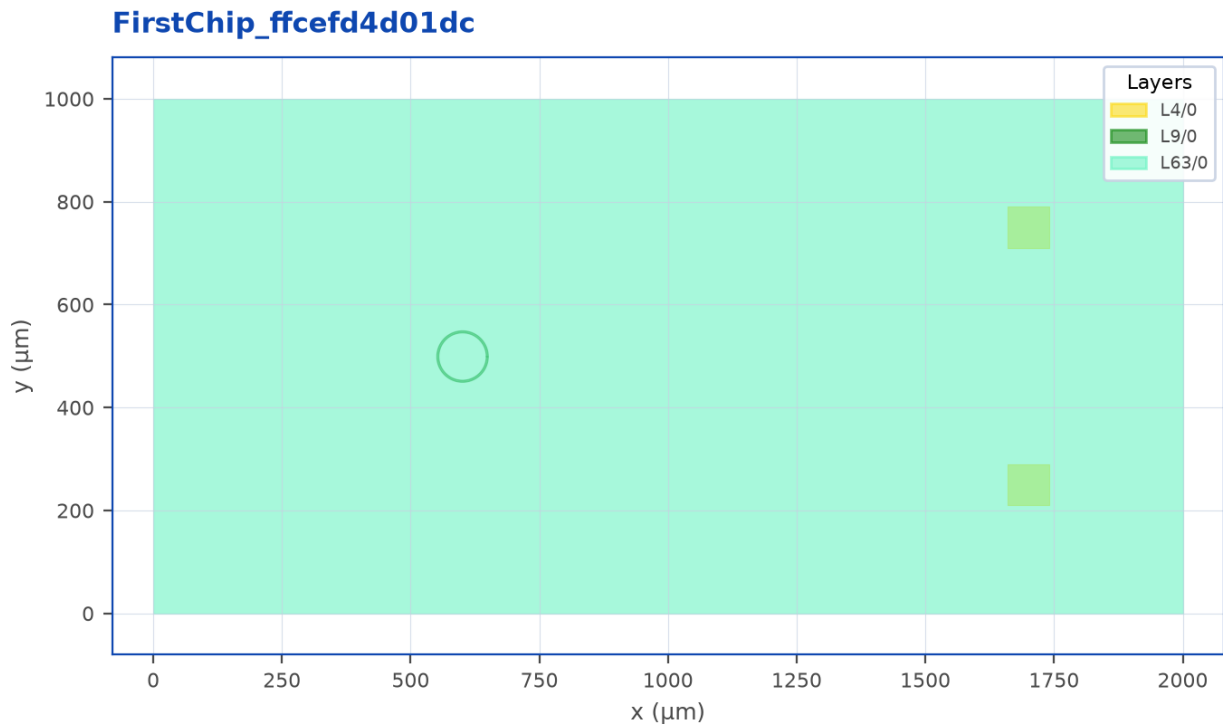


Figure 1.2.: Output of 02_first_chip.py.

Three patterns to internalize from this example:

1. **Define, add, place.** Build a shape, hand it to the cell with `c.add(...)`, then locate it with `c.Place(...).at(...)`.
2. **Capture the handle when you need multiple copies.** `pad_top = chip.add(pad)` returns a fresh handle for that specific instance. Each `chip.add(pad)` creates a distinct placed object. Keep both handles to move the two placements independently.
3. **.using("SW") overrides the default placement anchor.** By default `at((x, y))` puts the *bbox center* at `(x, y)`. The frame uses `using("SW")` to anchor at the south-west corner instead, so the chip spans `(0, 0)` to `(2000, 1000)` rather than centering on the origin.

1.4 What you just used

Symbol	What it is
<code>lm.CELL("X")</code>	A new cell named "X". Top-level container.
<code>lm.Rectangle</code>	A rectangular polygon shape.
<code>lm.Ring</code>	A ring (annulus) shape.
<code>lm.LayerSpec</code>	A (layer, datatype) pair.
<code>c.add(...)</code>	Adds a shape, port, bookmark, note, or sub-cell.
<code>c.Place(...)</code>	Returns a placement chain for an <code>add()</code> 'd thing.
<code>c.inspect()</code>	Prints a design summary; <code>as_dict=True</code> returns structured data.
<code>.to_gds(path?)</code>	Writes GDS; use <code>c.to_oas(path)</code> for OASIS.

The next chapter formalizes these into a vocabulary you'll use everywhere.

1.5 Phoebe Guide and native circuit experiments

Choose **Help** → **Start Phoebe Guide** to open or resume an editable project using **Lumicron Demo PDK 0.1.0** and its separate validation package. Build one MZI, simulate it, generate its physical Layout, verify it, prepare its Registry and curate a Notebook record. The Demo platform is fictional and educational, not fabrication qualified.

The short **Product Tour** introduces the application on first use. Replay it from **Settings** → **Phoebe** → **Replay Product Tour**. **Open Phoebe Guide** in the same Settings pane opens or resumes the separate engineering project; it never starts MZI instructions over an unrelated project. Existing project windows and unsaved work remain intact. Explicit Restart creates a fresh UUID-scoped copy. Choose a lesson from Lessons. Phoebe's spotlight follows real controls and objects; **Continue** always works, even when a target or result is unavailable. Engineering completion never advances the guide or turns skipped work into verified evidence. Drag the card's header to put Phoebe where you prefer. **Previous** changes guidance only. **Pause** leaves a small **Resume Guide** button; More actions offers Skip, Reference, Restart and Exit. Reopening retains the project's guide progress. **Restart** creates a fresh copy and preserves the previous project. Later-lesson restarts use a labeled ordinary MZI checkpoint. Legacy LearnLumicron resources and their machine identities remain readable; they are not silently converted to Demo designs.

The working project uses normal manifests, exact locks, Schematic, public Script, Layout, Registry and Notebook files. Local bundled dependencies have content digests; a Git dependency additionally has an exact revision. Refresh Dependencies reviews changes; **Keep Current** preserves the lock. No public Demo remote is implied. Optional Phase 18 recipes live under Reference/examples, outside the mandatory curriculum. The guide's short instructions link to this reference for detail.

Phoebe names the control and its location whenever she asks for an interaction. For example, **</> Generate Layout in the top toolbar** turns the logical circuit into editable physical Python and a Layout. A spotlight uses only a real mounted target; native title-bar controls outside its host retain clear directions with no guessed highlight. Command-K opens the real **Search / Commands** palette during guidance. Closing it leaves you on the same card; Continue is manual.

In Schematic, place **Source** and **Detector** from the normal IdeaKit Library and wire their terminals directly to the circuit. Select a Source or Detector and open **Properties** for its Basic/Advanced controls. The Source sets the wavelength sweep in micrometres, normalized amplitude and phase in degrees. One Source and multiple Detectors are supported. Detectors are non-loading observations; they create no physical device or fabricated geometry. Experiment settings remain separate from component parameters and the semantic circuit. This native workflow introduces no public Python Circuit, Source or Detector API.

Select a connection and open **Properties** to edit its routing target, profile, bends and propagation override. Valid picker/toggle choices commit directly; text and numbers commit on Return or focus change. Generate Layout, Run Simulation and Save first resolve the focused Inspector edit. Invalid input stays visible and blocks those actions until corrected or cancelled with Escape; there is no Apply step. Accepted edits use normal Undo/Redo. An explicit propagation override takes precedence. Otherwise an explicit route profile, or matching profiles on both canonical endpoint PORTs, supplies its declared propagation model. Different endpoint profiles require an explicit choice; width and layer alone do not determine phase.

- **Specified** uses positive authored interconnect target lengths and declared propagation models. Schematic drawing distances are never optical lengths.
- **Realized** uses qualified measurements from **Generate Layout**. That action executes a Lumicron-owned construction transaction against the exact PCells, validates that physical candidate, then publishes editable `from_schematic` Python and its Layout together. The same recorded public calls and resolved arguments drive construction and source emission. A failed candidate leaves the previous generated files intact and supplies no current measurements. No separate Measure Routes step is needed. Changed circuit inputs cannot reuse stale evidence or fall back to specified lengths. User-owned edited Python is not silently rebound to the generated circuit.

Automatic physical placement supports bounded co-facing parallel interfaces, including opposing wire draw order. It may orient/reflect components and choose spacing from authored targets. PDK bend limits and targets remain unchanged; infeasible routing still fails explicitly. This is not general feedback synthesis or fabrication qualification. To take over physical placement/routing, use the ordinary generated-script ownership workflow.

Successful native Generate Layout retains the last validated realization at each generated source path in `.lumicron/realizations`. Script **Outline** shows this historical origin separately from current Layout correspondence. Removing the managed header or editing the public calls does not

erase that saved evidence. Older generated files with private binding/verification calls remain runnable; new generated files do not contain those calls. Running public source independently does not recreate the managed transaction's semantic correspondence. It can leave the new artifact's bindings and circuit equivalence unresolved. Lumicron never matches edited instances by their names, order or geometry. The receipt is not a substitute for an admitted current sidecar, and older projects without a receipt do not acquire guessed history.

Symbol rotation and external-port body dragging change presentation only. Use the separate port terminal to draw a wire. Existing boundary-attached experiments and legacy instrument documents remain readable; the normal Library teaches directly wired Source/Detector objects.

Each attempt captures the semantic circuit revision, exact component bindings, parameters, experiment and resolved request. Completed, failed, cancelled and interrupted attempts remain in project-owned history under `.lumicron/circuit-simulation`. Inspect transmission, normalized output power and phase from the selected historical run. **Publish to Notebook** is deliberate; automatic run history and curated Notebook evidence are separate.

The plot adapts to the Simulation tray's available space; captured inputs and diagnostics remain under **Run details**. Click the plot or enter a wavelength in micrometers to select a scalar measurement. Numerical selections such as 1.31060 retain that exact wavelength. Between captured samples the displayed quantity is linearly interpolated and labeled **interpolated**; this does not rerun SAX or create a solver sample. Values outside the captured range are not extrapolated. With no selection, Notebook receives the complete run without an invented middle-wavelength measurement. A selected measurement is published with its trace, quantity, wavelength, value, evaluation method and captured run; subsequent selection changes do not alter that Notebook record.

The initial scope is linear optical wavelength-domain simulation using declared SAX compact models. Missing packages, absent models, unsupported PORT interfaces, invalid parameters and stale realized routes fail explicitly. Executing a supplied model does not establish its fabrication accuracy. Realized hierarchy combinations without qualified route measurements remain unavailable. Existing legacy instrument circuits retain their compatibility workflow; they are not the canonical experiment model taught here.

1.6 Code → Schematic

Use `cell.to_schematic()` after authoring a CELL's explicit component graph:

```
import lumicron as lm
import example_oband.all as oband

cell = lm.CELL("GuidePair")
a = cell.add(oband.Guide())
```

```

b = cell.add(oband.Guide())
cell.Place(b).at((100, 0))
cell.Route(a.ports["o2"], b.ports["o1"])
a.promote({"o1": "input"})
b.promote({"o2": "output"})
created = cell.to_schematic()
print(created.path)

```

The result is a frozen record with path (`pathlib.Path`), `circuit_id`, `name`, `sha256`, and a tuple of diagnostics. By default a new `.lumsch` is created under `Circuits/Generated` in the working project. An optional path selects a new `.lumsch` destination; an existing document is never overwritten.

Default filenames use the cell name and a numeric suffix when occupied; the filename is presentation, never semantic identity. For a PCell, its declared factory name supplies the engineering label instead of the generated geometry's parameter-hash name.

Each invocation creates a new circuit and new semantic occurrence/connection identities. Reopening the saved document retains its identities. This does not synchronize edited Python with an earlier Schematic.

Immediate eligible child instances become separate components. Reused subcells retain enterable definitions. Typed Route endpoints, paired Join endpoints and explicit promotion aliases establish connections. Raw polygons, raw GDS interiors, layout-only PCells (`schematic=False`) and ARRAY geometry do not become inferred components or wires; coverage diagnostics identify omitted content. Missing typed endpoints or a root PORT with no established relationship cause an explicit error. Unconnected child PORTs remain unconnected. Physical extraction and LVS are separate.

When a placed PCell's explicit Join, Route or promotion reaches an immediate `schematic=False` child, that PCell remains one atomic component with its declared external PORTs. Its validated internal endpoint evidence is retained, while hidden implementation children stay hidden. Other reusable circuit definitions remain enterable. This boundary does not invent a transfer model or conceal dangling endpoints; unresolved authored evidence still fails explicitly.

Exact package bindings remain attached where available. Without such a binding, the saved interface remains inspectable but a same-named library entry is never substituted. Schematic drawing coordinates are a deterministic presentation and carry no physical placement authority. Retained targets/profile/style/bend settings remain intent; measured physical length is not invented as a target. Other routing features are not automatically converted into new Schematic capabilities.

In Lumicron Script, publication waits until the entire run succeeds. Script stays active and a **Schematic created** toast offers **Open Schematic**. The action opens that exact document; dismissing it leaves Script active. A failed run publishes no conversion and offers no success action. Headless Python writes the document when the call succeeds, returns the same result and requires no GUI.

CHAPTER 2

Core concepts

Cells, geometry, ports, and placement handles describe the design. Bookmarks and Notes add application metadata.

Noun	What it is
LayerSpec	A (layer, datatype) pair, optionally with design rules.
Layer	A LayerSpec carrying min-width / min-spacing / min-radius.
Cell	A reusable hierarchical block, built with <code>lm.CELL</code> .
Port	A named connection point on a cell.
Bookmark	A named bbox bookmark - surfaces in the viewer's Bookmarks tab.
Note	User-authored information for Lumicron Notebook.

Verb	What it does
add	Adds a shape, port, bookmark, sub-cell, or array to a cell.
Place	Locates an added thing - chained with <code>at</code> , <code>move_by</code> , <code>rotate_by</code> .
Route	Connects two ports - auto by default, manual via <code>.go()</code> .

2.1 Naming convention

The library uses three case styles to make intent visible at a glance:

- **ALL CAPS** for language primitives: `CELL`, `ARRAY`, `PORT`.
- **PascalCase** for shapes, components, and PDK objects: `Rectangle`, `Ring`, `Waveguide`, `Pad`.
- **lowercase** for chain methods: `.at()`, `.go()`, `.style()`, `.add_bookmark()`; directional flips are `.flipX()` and `.flipY()`.

Once you've seen this once, scripts read like English sentences.

2.2 Layers

A layer is a (layer_number, datatype) pair. You can allocate one by hand:

```
metal = lm.LayerSpec(layer=4, datatype=0)
silicon = lm.LayerSpec(layer=9, datatype=0)
```

...or pull them from a PDK module, where they come pre-named with design rules attached:

03_pdk_layers.py

```
"""Using a PDK's predefined layers and design rules.

Instead of allocating layer numbers by hand, import a PDK module and
read the named layers from its `LAYER` namespace. Each entry is a
`Layer` (subclass of `LayerSpec`) carrying min-width, min-spacing, and
min-radius rules – the router and tapers consult those automatically.
"""

import lumicron as lm
import lumicron.pdks.elyon_demo.all as pdk

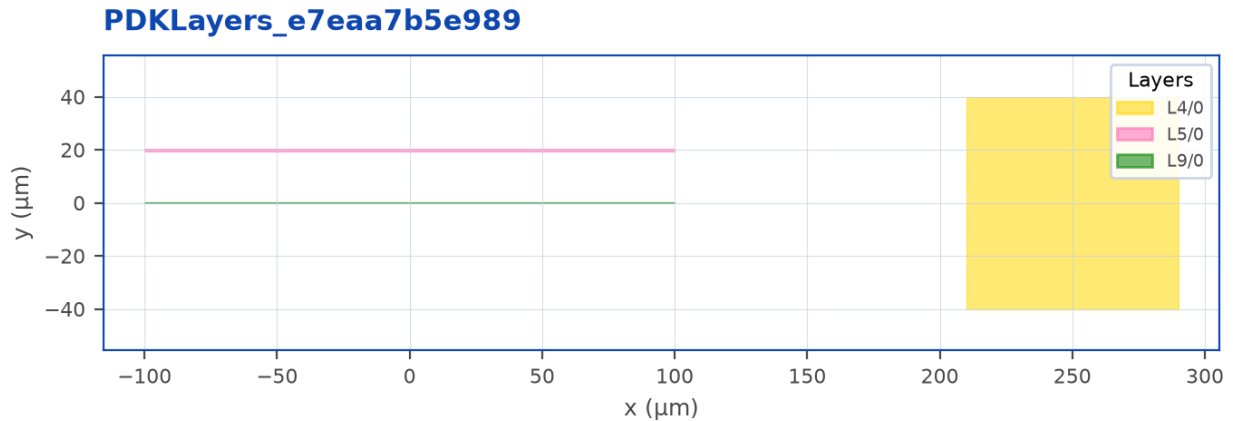
@lm.pcell
def PDKLayers():
    c = lm.CELL("PDKLayers")

    # Silicon waveguide-width strip on the SILC layer.
    strip = lm.Rectangle(x_dim=200, y_dim=0.5, layer=pdk.LAYER.SILC)
    c.add(strip)
    c.Place(strip).at((0, 0))

    # Silicon nitride strip, twice as wide, sitting above.
    siln = lm.Rectangle(x_dim=200, y_dim=1.0, layer=pdk.LAYER.SILN)
    c.add(siln)
    c.Place(siln).at((0, 20))

    # Metal-1 pad.
    pad = lm.Rectangle(x_dim=80, y_dim=80, layer=pdk.LAYER.MTL1)
    c.add(pad)
    c.Place(pad).at((250, 0))
    return c

if __name__ == "__main__":
    PDKLayers().to_gds("pdk_layers.gds")
```

Figure 2.1.: Output of `03_pdk_layers.py`.

Tip

A PDK's layers are not just numbers - they're `Layer` objects carrying `min_width`, `min_spacing`, and `min_radius`. The router applies the resolved minimum-radius rule to rendered curvature. A declared Euler layout radius is not itself a minimum-curvature guarantee.

2.3 Cells

A cell is a named container. It can hold:

- **Shapes** (Rectangle, Polygon, Circle, Ring, ...)
- **Sub-cells** (other CELLS, including from a PDK)
- **Arrays** (ARRAY of a child cell on a regular grid)
- **Ports** (PORT - labeled connection points)
- **Bookmarks** (Bookmark - labeled fit-to-screen bookmarks)

```
chip = lm.CELL("MyChip")
chip.add(lm.Rectangle(x_dim=100, y_dim=50, layer=metal))
```

`c.add(...)` returns a *handle*. Pass it to `c.Place()` to position that object. Sub-cell handles expose `handle.ports["o1"]`. Compact array occurrences expose geometry and bounds, not ports; use individual placements when each component needs a connection. For shapes you can pass either the original object or the handle to `Place()`; both work. Metadata additions return no geometry handle.

2.4 Ports

A port marks a directional connection point: a position, an angle, a width, and the layer it lives on.

```
chip.add(lm.PORT(
    "in", position=(0, 0), direction=0,
    width=0.5, layer=pdk.LAYER.SILC,
))
```

- `direction` is degrees CCW from +X (0 = East, 90 = North, 180 = West, 270 = South).
- `port_type` defaults to "optical". Pass `port_type="electrical"` for a metal pad's port.
- `route_profile=` (optional) attaches a cross-section recipe - see Chapter 5.

Access ports on a cell via `c.ports["name"]`; on a child handle via `handle.ports["name"]`.

2.5 Inspection and physical export

Keep the CELL as the authoring object. Lumicron Run discovers top-level cells; standalone scripts can export explicitly:

```
print(chip.inspect())
info = chip.inspect(as_dict=True)
chip.to_gds("chip.gds")
chip.to_oas("chip.oas")
netlist = chip.extract_netlist()
```

`inspect()` resolves a snapshot and reports definition-local objects, ports, bounds, route lengths with their measurement method and coverage, and custom data. It does not infer a path length from arbitrary polygons or certify fabrication/connectivity. The extracted netlist is a result, not an LVS pass.

`to_gds()` defaults to `max_points=0` (unlimited vertices). Use `max_points=199` only when the submission rules require legacy GDS limits. `.lum` persistence is owned by the application; there is no generic Script `save()` or `export()` verb. Internal Layout/compiler models are not designer-root constructors.

2.6 Cell information

Use `add_info()` for information your team wants to keep with a cell definition:

```
chip.add_info({"owner": "Photonics Team", "banana": 47,
              "path_length_um": 2345.67})
info = chip.inspect(as_dict=True)
print(info["cells"][chip.name]["user_info"])
```

Keys have no reserved meaning. `path_length_um` is your supplied value; it does not change a route length, DRC result or simulation. Repeated calls replace matching keys and retain other keys. Nested values are copied. Every placement of the definition shares its information; the method does not annotate individual instances. Existing `add_data()` remains available for its prior consumers.

Values may be strings, booleans, None, finite floats, integers from $-(2^{53} - 1)$ to $2^{53} - 1$, lists, and dictionaries with string keys, nested up to 32 levels. Unsupported values raise before changing the definition. The integer range supports exact exchange with the native application's JSON representation. Tuples, arbitrary objects, NaN and infinity are not converted.

Inspection keeps three separate per-cell records: `user_info` for these values, `gds_metadata` for original imported artifact records, and `lumicron_info` for bounded structural evidence. The Inspector uses the corresponding **User Information**, **GDS Metadata**, and **Lumicron Information** sections.

`CellFromGDS` retains source properties (including property bytes), labels, references and native PATH spines as inert source records. Their frame is the original source definition, even though the imported physical cell contains flattened polygons. A stored PATH's vertex-chain length excludes end extensions; it is not an optical length for the whole cell or a measurement of later edits. Polygon-only cells remain unresolved. None of this supplies `Route.length`.

Layout snapshots, .lum semantic records, and digest-matched GDS sidecars retain this information. Raw GDS and OAS exports carry physical geometry, not arbitrary user information. Importing a GDS without its matching sidecar cannot recover those user values. Selecting an individual exported definition recovers its information; flattening a parent does not promote each child's annotations onto that parent. Existing OAS viewer/geometry support does not imply a new `CellFromGDS` OAS authoring contract.

2.7 Package ownership

Design with `import lumicron as lm`. Author fabrication platforms with `import lumicron_pdk as lpdk`; write custom DRC/DFX libraries with `import lumicron_validate as lv`. These are separate public interfaces. The old `lumicron_api` namespace is a deprecated compatibility bridge.

Physical-simulation and experimental Script/Schematic APIs are archived, with internal application integrations retained. This guide does not prescribe their future replacement. XOR comparison is not a canonical Script primitive.

Part II.

Building Layouts

CHAPTER 3

Placement

Placement is how you position one cell, shape, or array within another. The pattern is always the same:

```
c.Place(thing).at((x, y))
```

`Place(...)` selects an already added object and returns a `PlacementChain`; selection alone does not move it. Positioning and transforms mutate that original handle. Keep the `c.add(...)` result when you need ports, bounds or promotion.

3.1 The chain

For a placed cell reference, these operations resolve as follows:

1. `at((x, y))` sets the target for the selected child anchor (the center by default).
2. `rotate_by(angle)` rotates counter-clockwise about the placed reference's center.
3. `move_by(dx, dy)` adds a shift in the parent cell's coordinates.
4. `flipX()` reflects in parent X (across Y); `flipY()` reflects in parent Y (across X). Flips preserve the selected placement anchor. Direct handle flips preserve the handle center.

04_placement_chain.py

```
"""The placement chain for cell references: at, move_by, rotate_by.

Every `c.Place(thing)` returns a chain. For references, `move_by` is a
parent-coordinate shift regardless of whether it precedes `rotate_by`.
"""

import lumicron as lm
import lumicron.pdks.elyon_demo.all as pdk
```

```

@lm.pcell
def PlacementChain():
    c = lm.CELL("PlacementChain")

    pad = lm.CELL("PlacementPad")
    pad.add(lm.Rectangle(x_dim=60, y_dim=60, layer=pdk.LAYER.MTL1))

    # 1. Plain placement.
    first = c.add(pad)
    c.Place(first).at((0, 0))

    # 2. Place + rotate 30°. The rotation is about the reference center.
    second = c.add(pad)
    c.Place(second).at((150, 0)).rotate_by(30)

    # 3. Place + rotate 90° + nudge by (10, 5) μm.
    third = c.add(pad)
    c.Place(third).at((300, 0)).move_by(10, 5).rotate_by(90)
    return c

if __name__ == "__main__":
    PlacementChain().to_gds("placement_chain.gds")

```

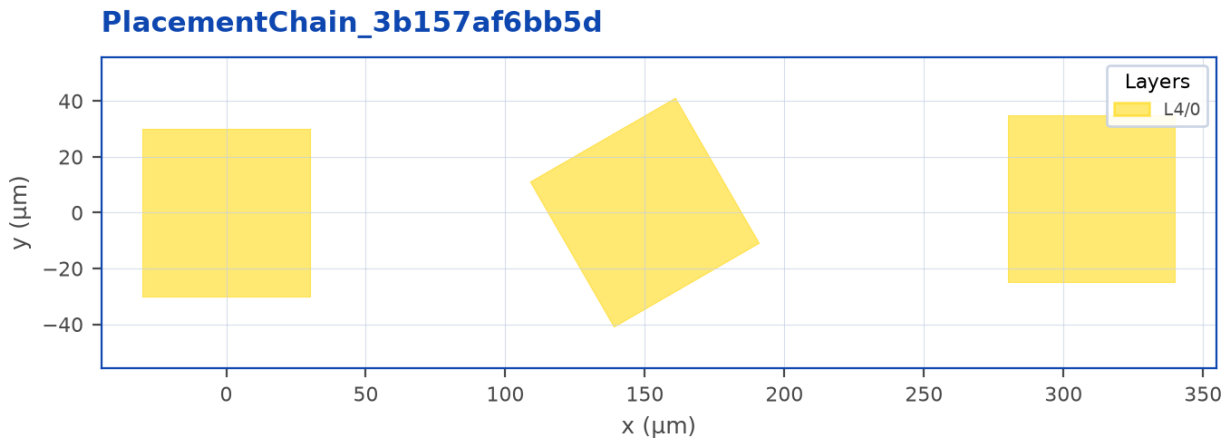


Figure 3.1.: Output of 04_placement_chain.py.

Note

For a cell reference, `move_by` accumulates and is applied after the placement target is resolved, regardless of whether you call it before or after `rotate_by`. The shift does not rotate with the reference. Rotations and reflections apply in call order for references, arrays, shapes, and groups; they do not commute. The stored GDS representation still applies local

reflection, rotation, then translation, but its coefficients encode the requested parent-axis operations.

3.2 Anchors

For sub-cells, you usually want to place by an edge or corner of the *child*, not the child's origin. `using(...)` switches the placement reference to a named anchor:

```
chip.Place(modulator).using("W").at((100, 50))
```

"W" (west), "E", "N", "S", "NE", "NW", "SE", "SW", and "C" are all valid. The cell's bbox supplies each anchor.

3.2.1 Selector, resolved point, or composite anchor

These three forms express different intentions:

```
c.Place(ref).using("SW").at((500, 0))
c.Place(ref).using(ref.SW).at((500, 0))
c.Place(ref).using(ref.ports["o1"], "S").at((500, 0))
```

- "SW" is a semantic selector. It resolves the mover's **current transformed physical bounding box**, including after a rotation or reflection. Cardinal anchors use the parent cell's axes at every angle.
- `ref.SW` is a resolved parent-coordinate point. `using(point)` captures its offset from the mover's current origin. It moves that captured point to the destination; it does not remember which object/property produced the tuple. Thus `using(other.SW)` deliberately translates the mover by `destination - other.SW`, leaving `other` unchanged. Later rotations do not turn that snapshot into a dynamic selector.
- `using(ref.ports["o1"], "S")` takes X from the mover's port and Y from its south bbox edge. The mover's own port is refreshed by name through transforms. Both coordinates reach the requested destination. Other supported selectors are bbox names, existing port names, and finite resolved points. Tuples are **not normalized UV coordinates**.

Selecting an anchor alone preserves the current position. With an existing destination, changing the anchor re-applies that destination to the new anchor. Oblique bounds use a private rendered-geometry snapshot; this can cost more than a cardinal bounds query.

A direct reference `rotate()` or `flipX()/flipY()` retains its established pivot: the transformed definition-box center. A fresh `Place(ref)` with no positioning constraints preserves that direct

behavior. Explicit using("C") selects the center of the current parent-axis physical bbox. For asymmetric oblique geometry those two centers can differ.

3.2.2 Differently sized MZIs

```
for i, delay in enumerate([40, 80, 140]):
    ref = c.add(PackagedMZI(delay=delay))
    c.Place(ref).using("SW").at((254.0 * i, 0.0))
```

Each MZI puts its own southwest bound on the row, despite different delay-arm heights. To put its input port on the column while preserving the same south edge:

```
c.Place(ref).using(ref.ports["o1"], "S").at((254.0 * i, 0.0))
```

The complete illustrative MZI, packaging, hierarchy and DOE scripts are in Examples/Phase8LayoutAPI/vnext. Their geometry is an API acceptance fixture, not an optically qualified component.

Waveguides and transition cells participate in the same contract. Their bbox is derived from the rendered path/reference geometry, including bend bulges, even though those factories build model elements directly rather than ordinary builder handles.

You can also align directly to another handle:

```
chip.Place(electrode).right_of(modulator) # electrode.W mates modulator.E
chip.Place(pad).above(electrode)         # pad.S mates electrode.N
```

Port names work as anchors too, and .at(...) accepts a port - so a placement can snap one port onto another:

```
chip.Place(wg).using("i1").at(bend.ports["o1"]) # translate: wg's i1 onto bend's o1
```

3.3 Joining ports

For the everyday case - “connect this port to that port, touching” - use Join. It moves the **first** port’s cell (translating *and rotating* it so the two ports face each other) onto the second port, which stays fixed:

```
bl = c.add(bend90)
wg = c.add(straight_wg)
c.Join(wg.ports["i1"]).to(bl.ports["o1"]) # wg moves; its i1 mates bl's o1
```

Read the moving endpoint from a placed handle returned by `c.add()`. The fixed endpoint can come from another handle in the same cell or from `c.ports["name"]` on that cell. Ports from another cell cannot be used as the fixed endpoint. A cell-definition port does not identify which instance should move. A single array element cannot be moved independently of its array.

`Join(a)` creates a `JoinChain` without moving anything. `.to(b)` refreshes both instance ports, moves the first instance, records the paired endpoints, and returns that original instance handle. A successful request completes once. Invalid ownership, array connectivity, or joining an instance to itself is rejected before mutation. A rejected target may be corrected and retried. The established `Join(a, b)` form remains supported with the same completed result.

`Join` is a butt-joint: no geometry is drawn. To connect ports at a distance with a routed waveguide, use `c.Route(...)`.

If the two ports disagree on width, layer, port type, or route profile, `Join` warns - that usually means the wrong ports are being mated - but it doesn't fail, since deliberate cross-section changes at a junction are legitimate.

3.4 Arrays

`ARRAY` repeats a cell on a regular grid. It emits a single AREF in GDS - far cheaper than placing `N` copies, and edits to the child propagate to every instance.

05_arrays.py

```

"""ARRAY - repeat a cell on a regular grid.

`lm.ARRAY(child, count, pitch)` creates a periodic array reference. It
emits a single AREF in GDS - far cheaper than placing N copies, and
edits to the child propagate to every instance.
"""

import lumicon as lm
import lumicon.pdks.elyon_demo.all as pdk

@lm.pcell
def Pad():
    """A single 80 x 80 μm metal-1 pad."""
    c = lm.CELL("Pad")
    pad = lm.Rectangle(x_dim=80, y_dim=80, layer=pdk.LAYER.MTL1)
    c.add(pad)
    c.Place(pad).at((0, 0))
    return c

@lm.pcell

```

```
def PadStrip():
    """1x8 linear pad array on a 127  $\mu$ m pitch."""
    c = lm.CELL("PadStrip")
    pad_array = lm.ARRAY(Pad(), count=8, pitch=(127, 0))
    c.add(pad_array)
    c.Place(pad_array).at((0, 0))
    return c

if __name__ == "__main__":
    PadStrip().to_gds("pad_strip.gds")
```

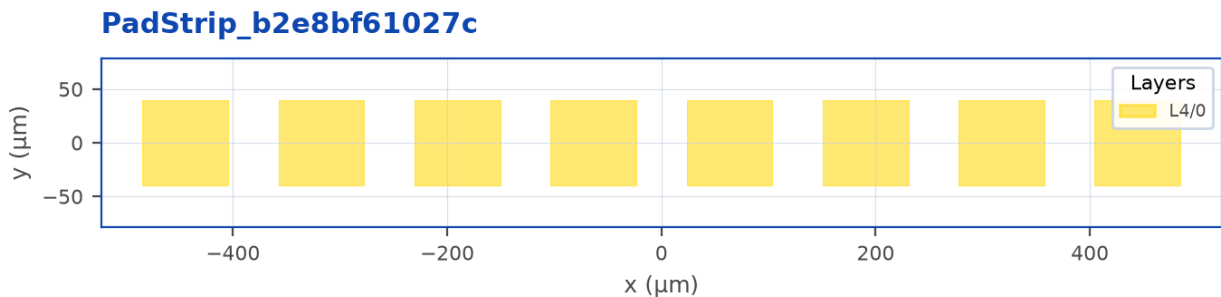


Figure 3.2.: Output of 05_arrays.py.

pitch is (dx, dy). Use scalar count=8 with (127, 0) for a horizontal strip or (0, 50) for a vertical column. For a rectangular grid use count=(nx, ny) and pitch=(dx, dy). A scalar count with both pitch coordinates nonzero is rejected. rotation= rotates each child; placement-chain transforms act on the entire compact array.

array[i] and array[column, row] inspect occurrence geometry/bounds. Occurrences have no public ports or promote() operation. Use separate c.add(child) placements for individually connectable devices. Historical serialized occurrence endpoints remain readable; new scripts do not manufacture them.

3.5 Hierarchy and re-use

Calling c.add(child_cell) multiple times creates multiple independent references (SRefs) - *not* duplicate geometry:

```
electrode_def = Electrode()           # define once

left = chip.add(electrode_def)         # SRef #1
right = chip.add(electrode_def)        # SRef #2
```

```
chip.Place(left).at((0, 0))
chip.Place(right).at((500, 0))
```

The GDS file contains *one* Electrode cell definition and two SRefs pointing at it. Edit `electrode_def`'s contents and both placements update.

3.6 Promoting child ports

If `Modulator` has ports `i1` and `o1`, and your top-level Chip wants to expose those for an outer cell to route into, use `handle.promote()`:

```
mod = chip.add(modulator_cell)
chip.Place(mod).at((0, 0))
mod.promote(ports=["i1", "o1"], prefix="mod_")
# Now chip.ports["mod_i1"] and chip.ports["mod_o1"] exist.
```

`promote()` accepts:

- A list of port names: `ports=["i1", "o1"]`
- A rename dict: `ports={"i1": "input", "o1": "output"}`
- None (default) - promotes everything.

`prefix=` is optional and lets you avoid name collisions when multiple children have ports of the same name.

`promote()` copies the selected **current** transformed ports onto the immediate parent. It is not a live constraint: finish placement first. Deeper interfaces require explicit promotion at each boundary; `promote` does not expose every descendant. Name collisions fail before copying any ports. Registry discovery recursively finds registered DUT occurrences within the supplied root, separately from port visibility; see the Registry chapter.

3.7 Positioning order and returns

All placement modifiers return the same chain. `using`, `at`, `relative_to`, `aligned_x`, `aligned_y`, and the relational operation occupy independent slots; repeated writes to a slot replace its previous value. There is one relational slot: the last of `above/below/left_of/right_of` wins.

The target starts at `relative_to(point) + at(offset)` when a reference is supplied; otherwise `at` is an absolute parent-cell destination. Relations supply defaults: `above` mates S to N and centers X, `below` mates N to S, `right_of` mates W to E and centers Y, and `left_of` mates E to W. Thus their default spacing is edge-to-edge zero, using current transformed bounds. `at` adds an offset

to a relational target; the relation's orthogonal centering overrides the offset on that axis. Explicit anchor/reference/alignment slots override relational defaults. `move_by` accumulates and is applied last.

3.8 Transform order

Transform order matters: `rotate_by` and `reflection` do not commute. The `at` destination and cumulative `move_by` offset are reapplied after each transform. With a semantic selector and the same final transform, `using("SW").at(target).rotate_by(angle)` and `rotate_by(angle).using("SW").at(target)` have the same final `bbox` constraint. A captured resolved point does not have that promise.

CHAPTER 4

Routing

Routing connects two ports with a waveguide. The auto-router handles the common cases; manual `.go()` chains let you take control where you need to.

4.1 Auto-routing

The simplest call:

06_ports_and_route.py

```
"""Ports and your first route.

A port is a named connection point on a cell. The auto-router can
connect any two ports – it picks Manhattan or curved geometry based on
the port directions, with bend radius drawn from the route layer's
design rules.
"""

import lumicon as lm
import lumicon.pdks.elyon_demo.all as pdk

@lm.pcell
def PortsAndRoute():
    c = lm.CELL("PortsAndRoute")

    c.add(lm.PORT(
        "in", position=(0, 0), direction=0,
        width=0.5, layer=pdk.LAYER.SILC,
    ))
    c.add(lm.PORT(
        "out", position=(400, 200), direction=180,
        width=0.5, layer=pdk.LAYER.SILC,
    ))
```

```

c.Route(c.ports["in"], radius=10).to(c.ports["out"])
return c

if __name__ == "__main__":
    PortsAndRoute().to_gds("ports_and_route.gds")

```

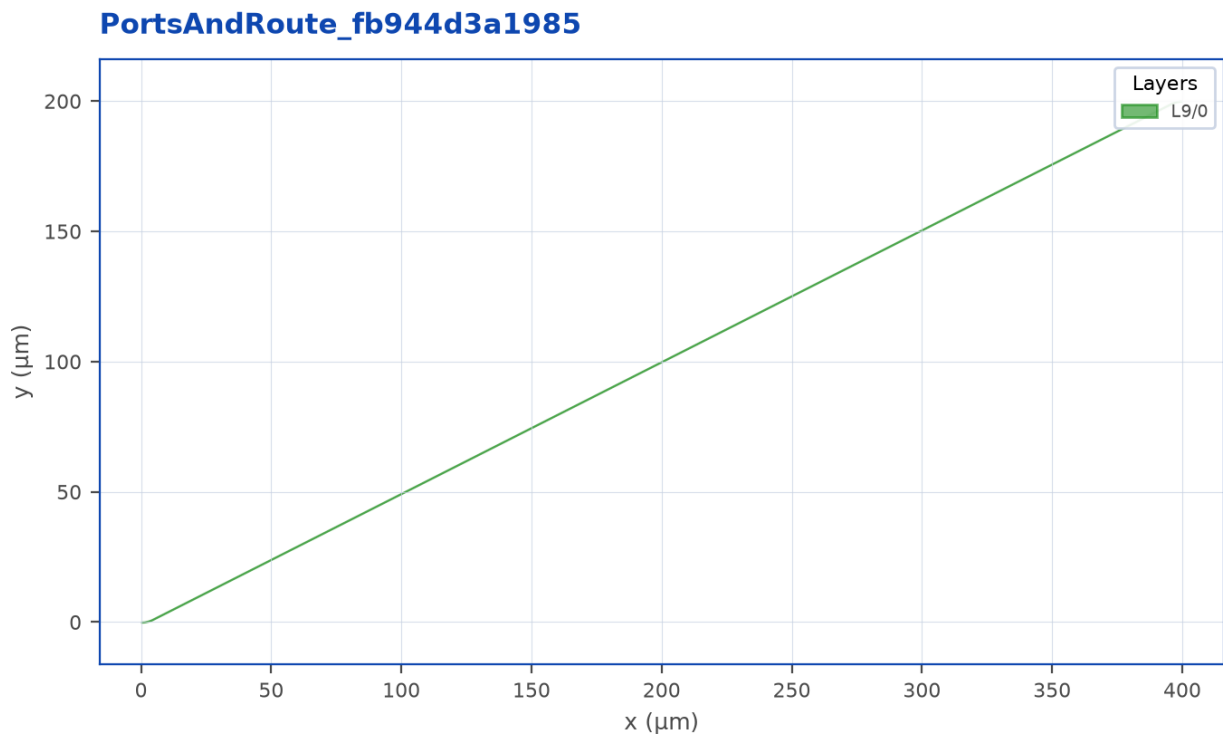


Figure 4.1.: Output of 06_ports_and_route.py.

`Route(a, radius=10).to(b)` picks geometry based on port directions:

Port pair	Geometry
Cardinal, perpendicular	Manhattan elbows
Cardinal, parallel + offset	Direct (Dubins arcs)
Any non-cardinal port	Direct

Force a specific style with `style=`:

07_route_styles.py

```

"""Three ways the auto-router can connect two ports.

`Route(a, b)` picks geometry based on port directions:
- cardinal + perpendicular → Manhattan (90° elbows)
- cardinal + parallel      → direct (Dubins arcs)
- any non-cardinal         → direct
You can force a style with `style="manhattan" | "direct" | "sbend"`.
"""

import lumicron as lm
import lumicron.pdks.elyon_demo.all as pdk

@lm.pcell
def RouteStyles():
    c = lm.CELL("RouteStyles")

    # Three port pairs stacked vertically. Same span, different styles.
    rows = [
        ("manhattan", 0, 200),
        ("direct", 0, 600),
        ("sbend", 0, 1000),
    ]
    for style, y, _ in rows:
        c.add(lm.PORT(
            f"{style}_in", position=(0, y), direction=0,
            width=0.5, layer=pdk.LAYER.SILC,
        ))
        c.add(lm.PORT(
            f"{style}_out", position=(400, y + 80), direction=180,
            width=0.5, layer=pdk.LAYER.SILC,
        ))
        c.Route(
            c.ports[f"{style}_in"],
            radius=20,
            style=style,
        ).to(c.ports[f"{style}_out"])
    return c

if __name__ == "__main__":
    RouteStyles().to_gds("route_styles.gds")

```

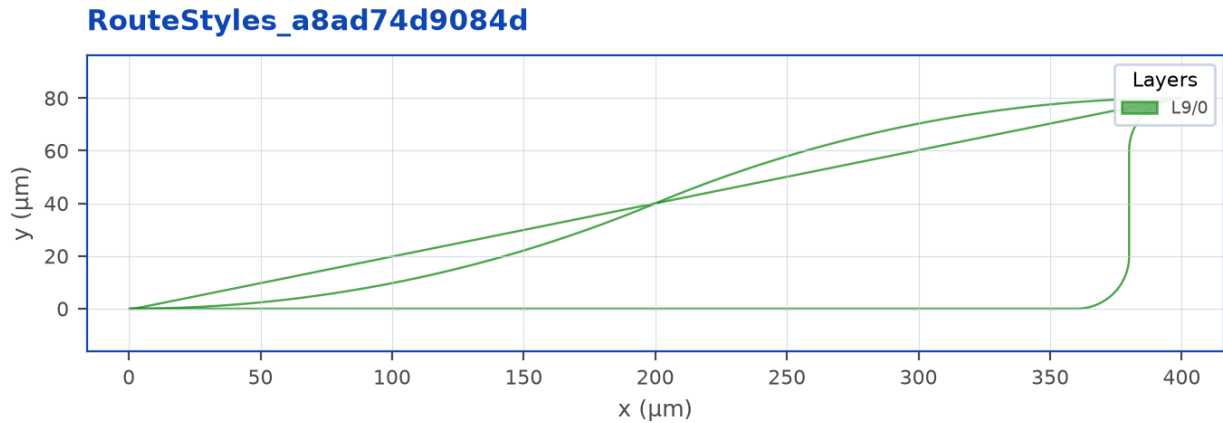


Figure 4.2.: Output of 07_route_styles.py.

i Note

radius sets the analytic bend layout footprint in μm . The required minimum curvature radius is checked on rendered geometry; Euler footprint and minimum curvature differ. Invalid geometry can fail during finalization. For straight-only routes (cardinal, no offset), pass radius=0.

4.2 Bends

Bend type is set on the `Route(...)` call:

```
chip.Route(a, radius=10, bend="circular").to(b) # constant-radius arc (default)
chip.Route(a, radius=10, bend="euler").to(b)    # clothoid - zero curvature at
↪ entry/exit
chip.Route(a, radius=10, bend="euler", p=0.5).to(b)
chip.Route(a, radius=10, bend=my_pcell).to(b)   # custom 90° PCell
```

The `p` parameter controls the Euler contribution. `p=0` is circular and `p=0.5` is fully Euler. The declared layout radius is not necessarily the minimum curvature radius; minimum-radius rules apply to the rendered curvature.

A custom bend is just a `CELL` with `i1` and `o1` ports - see the section-transitions example in Chapter 5.

4.3 Manual routing with `.go()` and `.jog()`

When you need a specific shape, take over with `.go()`:

08_manual_routing.py

```

"""Manual routing with .go() and .jog().

Sometimes you want explicit GPS-style turn-by-turn instructions:
- .go(direction, by=...) advances the route by that amount.
- .go(direction, to=...) advances until reaching that coordinate.
- .jog(direction, by=...) inserts a U-shaped detour offset.
Use cases: delay lines, MZI arms, anywhere you need a precise shape.
"""

import lumicron as lm
import lumicron.pdks.elyon_demo.all as pdk

@lm.pcell
def ManualRouting():
    c = lm.CELL("ManualRouting")

    c.add(lm.PORT("in", position=(0, 0), direction=0,
                  width=0.5, layer=pdk.LAYER.SILC))
    c.add(lm.PORT("out", position=(800, 300), direction=180,
                  width=0.5, layer=pdk.LAYER.SILC))

    # Manual chain: east 200, north 300, east to port_b, then auto-finish.
    (c.Route(c.ports["in"], radius=15).to(c.ports["out"])
     .go("E", by=200)
     .go("N", by=300)
     .go("E", to=c.ports["out"].position[0] - 50))
    return c

if __name__ == "__main__":
    ManualRouting().to_gds("manual_routing.gds")

```

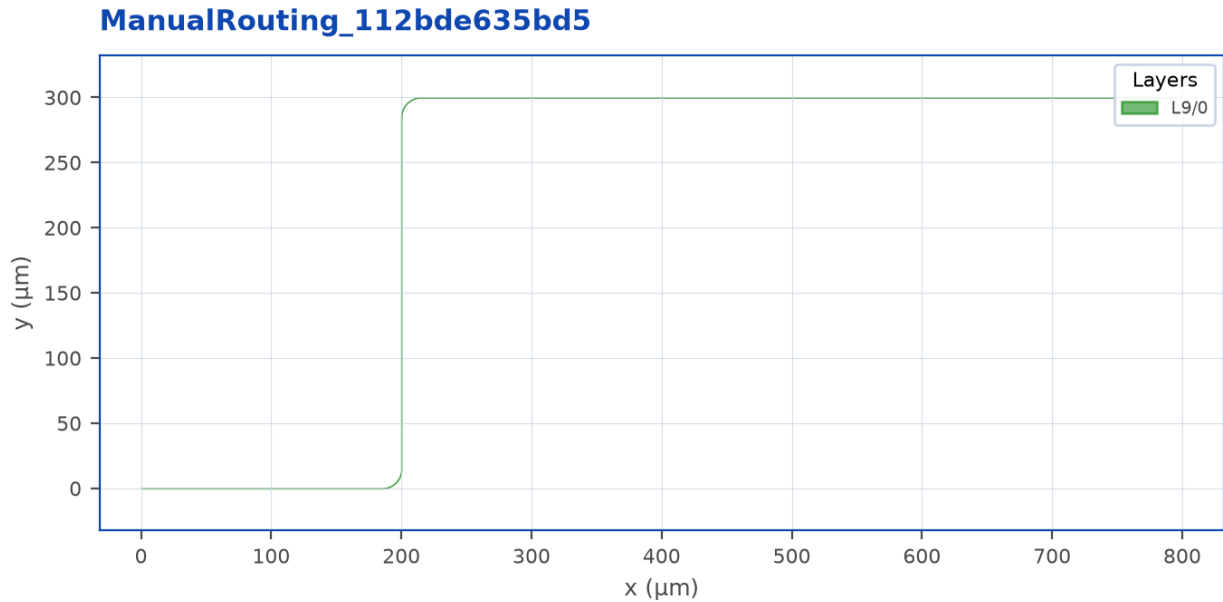


Figure 4.3.: Output of 08_manual_routing.py.

- `.go(direction, by=...)` advances the unrounded waypoint by that many μm . A bend trims the adjoining straight: a north waypoint 50 μm away with a 5 μm circular corner leaves 45 μm of vertical straight. The waypoint displacement is not a requested straight length.
- `.go(direction, to=...)` advances until the named coordinate.
- `.go("NE" / "NW" / "SE" / "SW", by=(dx, dy))` lays an S-bend in that diagonal.
- `.jog(direction, by=...)` inserts a U-shaped detour shifting the route by `by`. Useful for delay lines and MZI arms.

After your last `.go()`, the route auto-finishes to `port_b` if not already aligned.

4.4 Routing bundles

To route many parallel channels with a single call:

10_route_bundle.py

```
"""Route - route N parallel channels with one call.
```

```
Three modes, picked by what you pass:
```

- | | |
|--|---|
| - <code>`Route(...)`</code> (default style) | → direct: per-channel,
no coordination. |
| - <code>`Route(..., style="manhattan")`</code> | → auto-spined manhattan
bundle; the router |

```
- `Route(...).go("E", by=...)`
```

picks the shortest
path from ports_a's
centroid to ports_b's.
→ manhattan-spined too,
but you steer the
spine leg by leg.

This example uses the auto-spined form, which is the right default when
your bundle just needs to "go from A to B."
"""

```
import lumicron as lm
import lumicron.pdks.elyon_demo.all as pdk

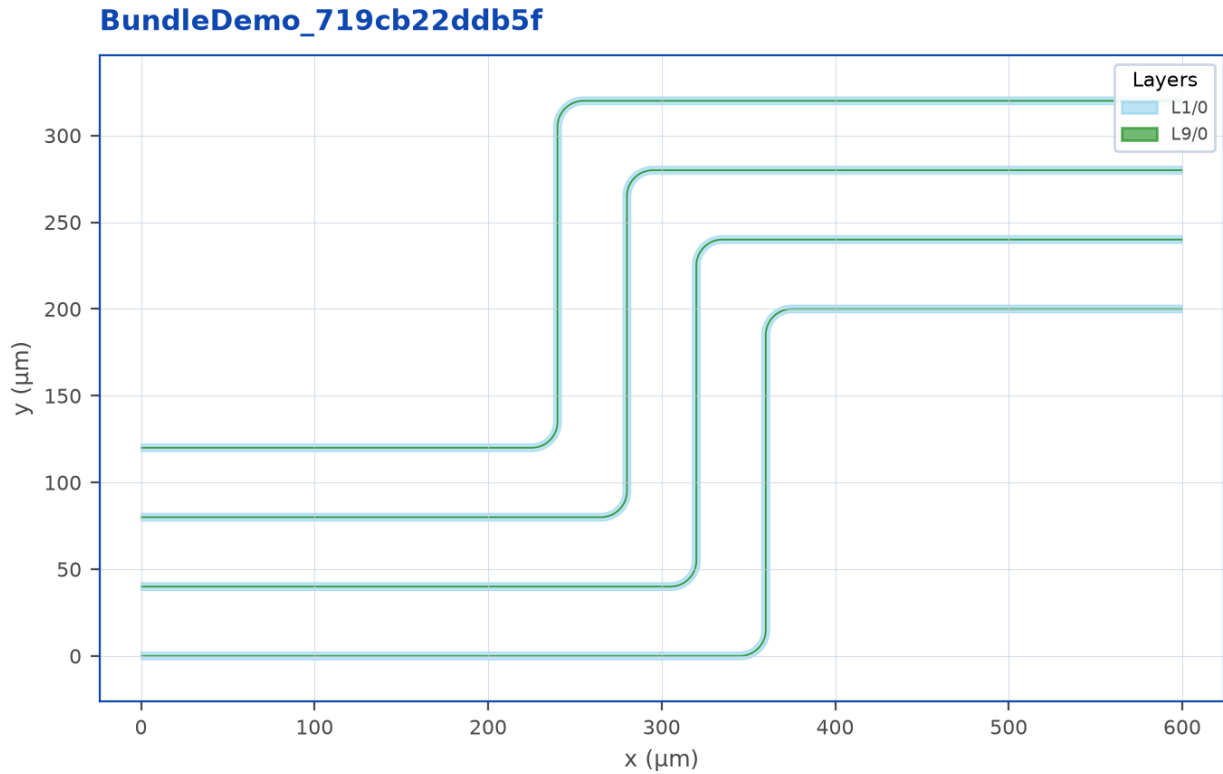
@lm.pcell
def BundleDemo(n: int = 4, pitch: float = 40.0, span: float = 600.0,
               y_shift: float = 200.0):
    c = lm.CELL("BundleDemo")

    for i in range(n):
        y = i * pitch
        c.add(lm.PORT(
            f"in{i}", position=(0, y), direction=0,
            route_profile=pdk.RP.silc_strip,
        ))
        c.add(lm.PORT(
            f"out{i}", position=(span, y + y_shift), direction=180,
            route_profile=pdk.RP.silc_strip,
        ))

    ports_a = [c.ports[f"in{i}"] for i in range(n)]
    ports_b = [c.ports[f"out{i}"] for i in range(n)]
    c.Route(ports_a, spacing=pitch, radius=15,
            style="manhattan").to(ports_b)

    return c

if __name__ == "__main__":
    BundleDemo().to_gds("bundle_demo.gds")
```

Figure 4.4.: Output of `10_route_bundle.py`.

`Route(ports_a, spacing=10, radius=15).to(ports_b)` returns a chain that broadcasts to every channel. There are three modes:

- **Direct mode** (default - no `style=`, `.go()` or `.jog()`): each pair routes independently. The bundle is a convenience grouping; channels don't coordinate. Use this when ports are already aligned and you just want a vectorized `Route()`.
- **Manhattan-spine, auto** (`style="manhattan"`, no `.go()` or `.jog()`): the bundle auto-computes a shortest-path manhattan spine from `ports_a`'s centroid to `ports_b`'s centroid - 1, 2, or 3 legs depending on the geometry. Channels share the spine, fanouts absorb pitch differences at each end, and the bundle's `spacing=` is honored everywhere. This is the right call for almost every "I want N parallel waveguides on a fixed pitch" scenario.
- **Manhattan-spine, manual** (`.go(...)` or `.jog(...)`): drive the spine yourself, leg by leg. Use this when the auto-spine would clip an obstacle or you need a non-shortest route.

Access individual channels through `bundle.routes`, for example `bundle.routes[0]`. The bundle itself is not iterable or indexable. Use `.taper(...)` to broadcast tapers. For a bundle that has not been steered with `.go()` or `.jog()`, `.match_lengths(...)` rebuilds every channel as an independent trombone (see *Bundle PLM* below). Exact length matching on a manually steered shared spine is unsupported and raises an error.

Warning

A manhattan bundle's spine has minimum-leg-length constraints set by the bundle's *width* $((n_channels - 1) \cdot spacing)$ and bend radius. If `ports_a / ports_b` aren't far enough apart to fit those staircase corners, you'll get an explicit error telling you the geometry is too tight. Either widen the perpendicular gap, reduce spacing, reduce radius, or use direct mode if collisions are acceptable.

4.5 Length matching

Two parallel routes don't have the same physical length unless you ask. `.match_length_to(other)` adds a single U-meander to the shorter route so its total length equals the longer one:

11_length_matching.py

```

"""Length matching – equalize optical path lengths across channels.

Two arms with different physical paths. `match_length_to(other)` adds a
single U-meander to the shorter arm so its total length equals the
longer arm's length. Critical for MZI arms, parallel fiber arrays, and
any interferometer-style geometry.
"""

import lumicron as lm
import lumicron.pdks.elyon_demo.all as pdk

@lm.pcell
def LengthMatch(pitch: float = 50.0):
    c = lm.CELL("LengthMatch")

    for i in range(2):
        y = i * pitch
        c.add(lm.PORT(f"a{i}", position=(0, y), direction=0,
                      route_profile=pdk.RP.silc_strip))
        c.add(lm.PORT(f"b{i}", position=(800, y), direction=180,
                      route_profile=pdk.RP.silc_strip))

    # Bottom arm jogs south then back – makes it the longer reference.
    bot = (c.Route(c.ports["a0"], radius=15).to(c.ports["b0"])
           .jog("S", by=80))

    # Top arm: straight, then matches bot's length with an auto U-bend.
    (c.Route(c.ports["a1"], radius=15).to(c.ports["b1"])
     .match_length_to(bot))

    return c

```

```
if __name__ == "__main__":
    LengthMatch().to_gds("length_match.gds")
```

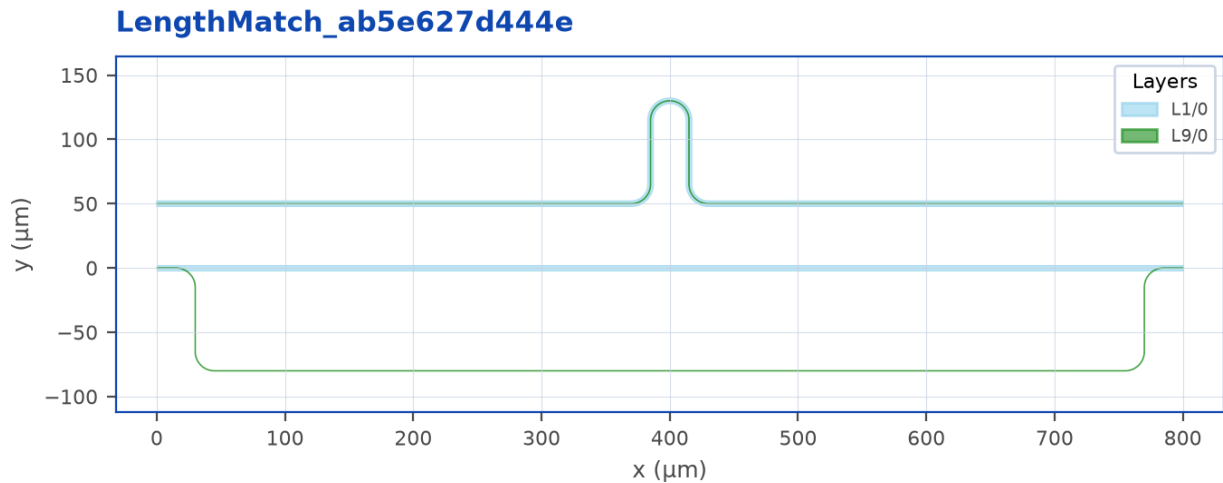


Figure 4.5.: Output of 11_length_matching.py.

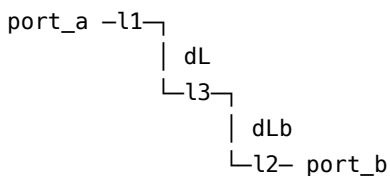
For an individual route, optional controls include:

- `delta=` - extra ΔL to add on top of the match (e.g. for a deliberate phase offset).
- `at=` - where along the route to insert the meander (cumulative distance from `port_a`).
- `span=`, `amplitude=`, `side=` - pin the meander's footprint instead of letting the matcher choose.

`bundle.match_lengths()` accepts only `target`, `delta`, `mirror`, and `tolerance`; individual-route footprint controls do not apply to bundles. `target_length=` on `Route` is a positive physical path-length target.

4.5.1 Bundle PLM - rectangular detour

When you call `.match_lengths()` on a bundle that hasn't been steered with `.go()` or `.jog()`, every channel is rebuilt as an independent rectangular detour:



Each channel's two perpendicular leg lengths (dL_k , dLb_k) are solved analytically so the **total path length** ends up identical across the bundle (plus an optional per-channel `delta=` offset). The math is closed-form: two equations in two unknowns per channel, no iteration. Channels with **different forward distances** (Lx_k) are supported - a fan-out from a fiber array on one side to spread-out ports on the chip face still equalizes correctly.

```
chip.Route(ports_a, ports_b, spacing=10, radius=15) \
    .match_lengths()
```

Each channel's detour verticals occupy a unique main-axis position (the algorithm packs them in monotonic order by sort index), so legs from different channels never share a column.

Routing direction is auto-detected from the first two `ports_a` positions: if they share an x , the bundle is a vertical column and routes go horizontally; if they share a y , horizontal row and routes go vertically. Pairs preserve the order of the supplied lists. Only `sort=True` requests transverse-coordinate pairing.

mirror= - flip the detour

By default the detours bulge in the $+trans$ direction. Pass `mirror=True` to flip every channel to the $-trans$ side, or pass a `list[bool]` to mirror per channel:

```
# Whole bundle bulges down instead of up
chip.Route(pa, pb, spacing=10, radius=15).match_lengths(mirror=True)

# Two channels up, two channels down - split bundle into stacks
chip.Route(pa, pb, spacing=10, radius=15) \
    .match_lengths(mirror=[False, False, True, True])
```

Mirroring is a pure geometric flip - channel lengths and net transverse displacements are preserved; only the bulge direction changes. Per-channel mirror is useful when obstacles split the routing into two halves. Adjacent channels going in opposite directions can collide if amplitudes are large; verify visually.

Warning

The detour needs main-axis room for $l1 + l3 + l2$ - the inner straight $l3$ must be at least $2 \cdot R$ to inscribe the inner bends. If your ports are too close together for the geometry to fit, you'll get a clear error naming the constraint. Move ports further apart or reduce `radius / spacing`.

Bundles steered with `.go()` or `.jog()` bypass this path. Exact length matching on their manually steered shared spine is unsupported.

4.6 Section-level overrides

A single Route can change profile, layer, bend, or width as it travels. The chain modifiers `.style()`, `.radius()`, `.bend()`, and `.width()` configure the *next* section, taking effect on the next `.go()` / `.jog()`. They never mutate the route's global style - pass "default" to revert.

09_section_transitions.py

```

"""Section-level chain modifiers – Lumicron's wedge feature.

A single Route can change layer, profile, bend type, and width as it
travels. The chain modifiers .style(), .radius(), .bend(), and
.width() apply to the next .go() and persist until you change
them. Pass "default" to revert to the Route()-time global.

Auto layer transitions slot in PCells (e.g. SiLN ↔ SiLC bridges) at
every section boundary that crosses layers – you don't drop them in
manually.
"""

import lumicron as lm
import lumicron.pdks.elyon_demo.all as pdk

@lm.pcell
def SectionTransitions():
    c = lm.CELL("SectionTransitions")

    c.add(lm.PORT(
        "in", position=(0, 0), direction=0,
        route_profile=pdk.RP.silc_strip,
    ))
    c.add(lm.PORT(
        "out", position=(800, 400), direction=180,
        route_profile=pdk.RP.silc_strip,
    ))

    (c.Route(c.ports["in"]).to(c.ports["out"]))
    # Section 1 – silicon strip (default from port_a.route_profile).
    .go("E", by=250)
    # Section 2 – switch to silicon-nitride strip.
    .style(profile=pdk.RP.siln_strip)
    .go("E", by=200)
    .go("N", by=150)
    # Section 3 – back to defaults but with a wider core.
    .style("default")
    .width(3)
    .go("N", to=c.ports["out"].position[1])
    .go("E", to=c.ports["out"].position[0]))
    return c

```

```
if __name__ == "__main__":
    SectionTransitions().to_gds("section_transitions.gds")
```

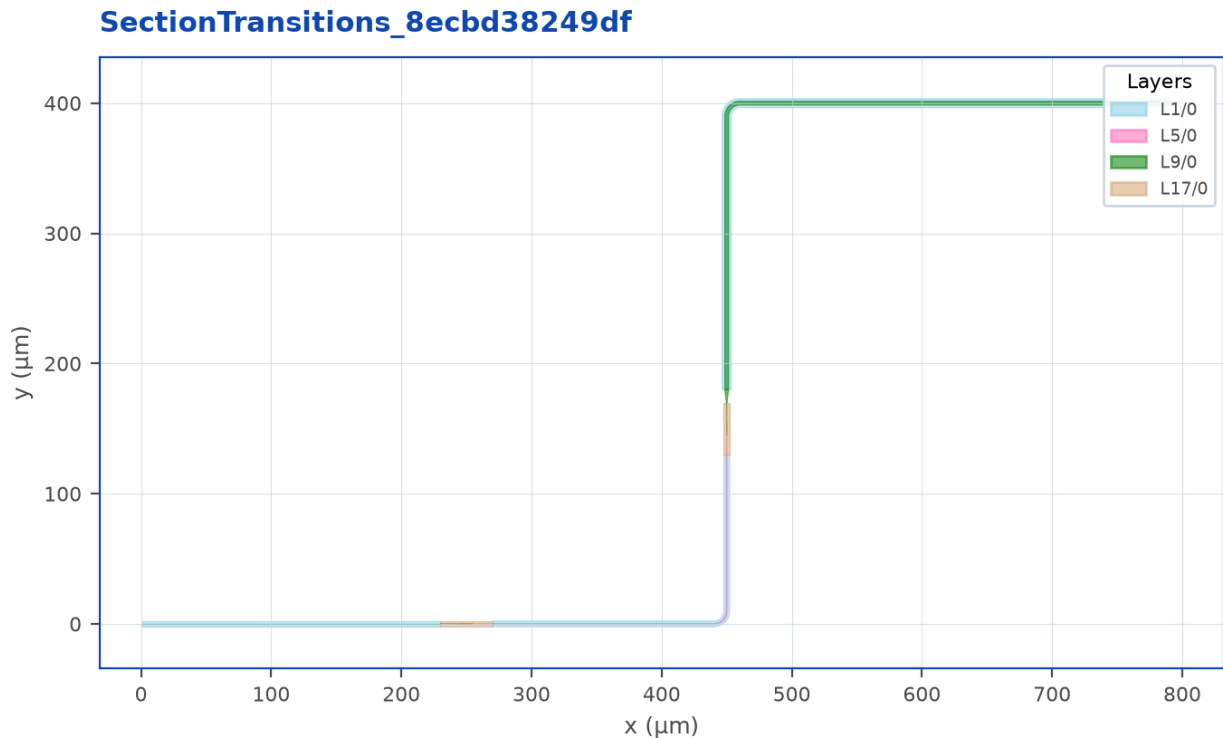


Figure 4.6.: Output of 09_section_transitions.py.

What's happening in the example:

1. **Section 1** - silicon strip, inherited from `port_a.route_profile`.
2. `.style(profile=pdk.RP.siln_strip)` - section 2 is silicon nitride, with an automatic layer-transition PCell at the boundary.
3. `.style("default")` - revert profile/layer/bend.
4. `.width(3)` - section 3 is back on silicon, but with a 3 μm core. The router auto-inserts a linear taper at the boundary.

Tip

This is the feature most users miss for longest. If you find yourself emitting two separate `Route()` calls and stitching them by hand to change layer mid-route, you want section-level transitions.

`.style()` is the umbrella - it accepts any subset of `profile=`, `layer=`, `radius=`, `width=`, `bend=` plus a positional argument that auto-dispatches by type:

<code>.style(arg)</code>	Effect
<code>.style(rp_obj)</code>	Switch route profile.
<code>.style(layer_spec)</code>	Switch the primary layer only.
<code>.style("name")</code>	Look "name" up in <code>cell.styles</code> .
<code>.style("default")</code>	Revert every slot to the <code>Route()</code> -time global.

`.radius(r)`, `.bend(b)`, `.width(w)` are sugar for the corresponding `style(...)` kwargs when you only need one knob.

4.7 Tapers

Tapers come in two forms - port-end and chained - and three placements:

```
chip.Route(a, b, radius=10) \
    .start_taper(width=2, length=10)           # at port_a
    .taper(offset=150, width=1, length=5)      # mid-route, 150 µm in
    .taper(offset=200, width=2, length=10)     # mid-route, 200 µm after that
    .end_taper(width=1, length=10)            # at port_b
```

- `start_taper` / `end_taper` configure the port-end transitions explicitly.
- `.taper(offset=, width=, length=)` inserts a positional taper. `offset` is measured from the end of the previous chain step (a `.go`, `.jog`, prior `.taper`, or `port_a`).

Width changes carry forward - the running body width after each taper is the taper's target. If your final width doesn't match `port_b.width`, the router auto-inserts a bridge taper at `port_b` and warns.

4.8 Layout-aware routing

`clearance` requests clearance around placed-cell bounding boxes for supported Manhattan routing. It is not a foundry DRC certificate or a general obstacle solver for every route style. Inspect the result and run the relevant checks for the actual PDK.

```
chip.Route(a, b, radius=10, style="manhattan", clearance=5)
```

4.9 Endpoint straights and loopbacks

`start_straight` and `end_straight` are minimum physical straight lengths in micrometres, measured between the port and the bend tangent, excluding the bend footprint. They apply to auto-routing, not to explicit `.go()` waypoints. A shared U-turn may lengthen one side to satisfy the other side's constraint or the port geometry. They do not set the total route length.

For two parallel, same-facing grating-coupler ports, a loopback can usually be expressed as:

```
c.Route(couplers.ports["GC0"], couplers.ports["GC1"],
        start_straight=50.0, end_straight=50.0)
```

The port route profile supplies the cross-section and default bend. Use `.go()` only when you need to specify the intervening path. For a parameterized loopback, place these lines inside an `@lm.pcell` factory and pass its length parameter to the endpoint straight settings. A PDK custom bend can have an arbitrary shape; do not infer circular length from its footprint.

4.10 Endpoint requests and compatibility

```
request = c.Route(a, radius=10)    # no geometry yet
route = request.to(b)              # RouteChain
route.width(1.0).go("E", by=60)   # existing section modifiers
```

Use `Route(source, options...).to(destination)` for new scripts. The two-endpoint `Route(a, b, options...)` remains supported. Source/destination may both be individual ports or nonempty equal-length lists/tuples. Mixed scalar/collection calls are rejected. A completed request returns the existing scalar or bundle chain and completes once; it does not add a new routing engine. Modifiers belong on the result **after** `.to()`.

Handle-derived endpoints must belong to immediate instances in the same parent cell and are refreshed when `.to()` runs. Explicit standalone `PORT` values retain their supplied coordinates and unresolved identity; a coordinate alone never proves an instance endpoint. Use parent-cell coordinates consistently. Arrays have no connectable occurrences. After routing, moving an endpoint does not make the route a live placement constraint; regenerate/recreate the route to reflect the new geometry.

The statement that creates the request supplies route source provenance, including when `.to()` is on a later line. Existing source-anchor continuity rules still apply: named unambiguous statements can survive sibling insertion, while repeated/ambiguous construction sites do not gain invented durable identity. Source-file movement and arbitrary Python rewrites are outside that guarantee.

4.11 PDK width permission and transitions

A profile's `port_width` is a **fallback** for a port whose width is omitted. The current router honors explicit endpoint widths; it does not widen two equal narrow ports to the nominal width automatically. Explicit section-width overrides can request a wider body.

PDK authors can share `lpdk.RoutingPolicy(auto_taper=False)` across their route profiles. The default is `True`, preserving generic linear tapers. If any participating profile forbids synthesis, a needed generic taper fails visibly at geometry resolution. Explicit compatible custom taper CELLS remain available through the existing `.taper(..., cell=...)`, `.start_taper(cell=...)`, and `.end_taper(cell=...)` options. They must declare exactly two physical ports with matching widths, layer, type, facing directions and requested straight span. Their origin is arbitrary; Lumicron aligns their ports and does not resize their geometry. Use matching widths or a PDK-provided transition when taper synthesis is forbidden. A supplied custom device is not fabrication qualification.

Finite positive widths/lengths are required. Endpoint `length=0/disabled=True` retains its explicit suppression behavior; it is not proof of physical width matching. Required endpoint/section tapers must fit their straight span, rather than silently disappearing. Positional tapers that overlap a bend retain the existing warning-and-skip contract; heed that warning. Multi-stroke profile taper changes retain the existing sibling-stroke warning and do not certify cladding transitions.

Distinct same-layer endpoint profile definitions are rejected: the layer-pair transition registry cannot establish strip-to-slot or other same-layer cross-section changes. Place a declared transition CELL explicitly and route to its separate ports. For different layers, the existing TransitionRegistry can supply a factory with `i1/o1` ports on the declared layers; automatic placement uses `transition_offset`. Nothing chooses a generic elevator without that declaration. Cross-layer routing returns the downstream sub-chain; the logical route group also owns the upstream geometry and transition. A fixed `target_length` across that transition remains unsupported.

4.12 Inspecting and completing routes

`RouteChain.length` and `c.inspect()` resolve private snapshots. They report established centerline length, including supported taper/bend decomposition, rather than a polygon-perimeter estimate. This can be expensive and can raise an unresolved routing error. GDS-backed custom bends without explicit centerline evidence do not gain an invented length. `inspect(as_dict=True)` retains coverage/method information; user metadata is not a measured simulation result.

Scalar modifiers remain `style`, `port_directions`, `radius`, `bend`, `width`, `match_length_to`, `taper`, `start_taper`, `end_taper`, `go`, and `jog`; `name` is a writable label, `length` a read, and `rules()` prints guidance. Bundle modifiers remain `width`, `radius`, `bend`, `style`, `port_directions`, `taper`, `match_`

`length_to`, `match_lengths`, `finalize`, `go`, and `jog`, with channels in routes. Their complete signatures are in the generated reference.

Pending style/width/radius/bend overrides are section state, not a global transformation of already committed geometry. A subsequent `go/jog` commits them; a trailing concrete override reconciles the final section. Repeating a pending value replaces it; "default" restores the Route-time setting where accepted. `taper(offset=...)` is measured from the last committed cursor, so chain order is meaningful. A scalar `go` with both `by` and `to` retains legacy by precedence; the bundle rejects that combination. Prefer supplying exactly one. Bundle manual steering still disables exact length matching.

Malformed manual steering/style calls restore their pending authoring state. Failed route creation restores newly added route/dependency/provenance records. Final geometry resolution uses an independent snapshot: a late failure does not finalize partial taper/bend geometry into the live authoring cell. An error does not mean a physically impossible request became valid; correct its constraints before export.

CHAPTER 5

Route profiles

A **route profile** is the cross-section recipe for a waveguide: which layers to stamp, what width each is, where periodic features (gratings, periodic claddings, slot rails, electrical stitching) sit. Once a profile exists, you stop thinking about per-stroke geometry and just point at the profile.

5.1 Using a profile

The demo PDK ships several ready-made profiles in `pdk.RP`. Attach one to a port and every Route from that port adopts it:

12_route_profiles.py

```
"""Route profiles – cross-section recipes for waveguides.

A route profile bundles everything that varies along a route: layers,
strokes, widths, periodic teeth (gratings, tapers, periodic claddings).
Attach one to a port and every Route from that port adopts it.
"""

import lumicron as lm
import lumicron.pdks.elyon_demo.all as pdk

@lm.pcell
def ProfileDemo():
    c = lm.CELL("ProfileDemo")

    # The PDK ships ready-made profiles in `pdk.RP`. Here we route with
    # the silicon-nitride strip – the profile stamps SiLN at the right
    # width and inherits its bend radius.
    c.add(lm.PORT(
        "in", position=(0, 0), direction=0,
        route_profile=pdk.RP.siln_strip,
    ))
```

```

c.add(lm.PORT(
    "out", position=(500, 200), direction=180,
    route_profile=pdk.RP.siln_strip,
))

c.Route(c.ports["in"], radius=15).to(c.ports["out"])
return c

if __name__ == "__main__":
    ProfileDemo().to_gds("profile_demo.gds")

```

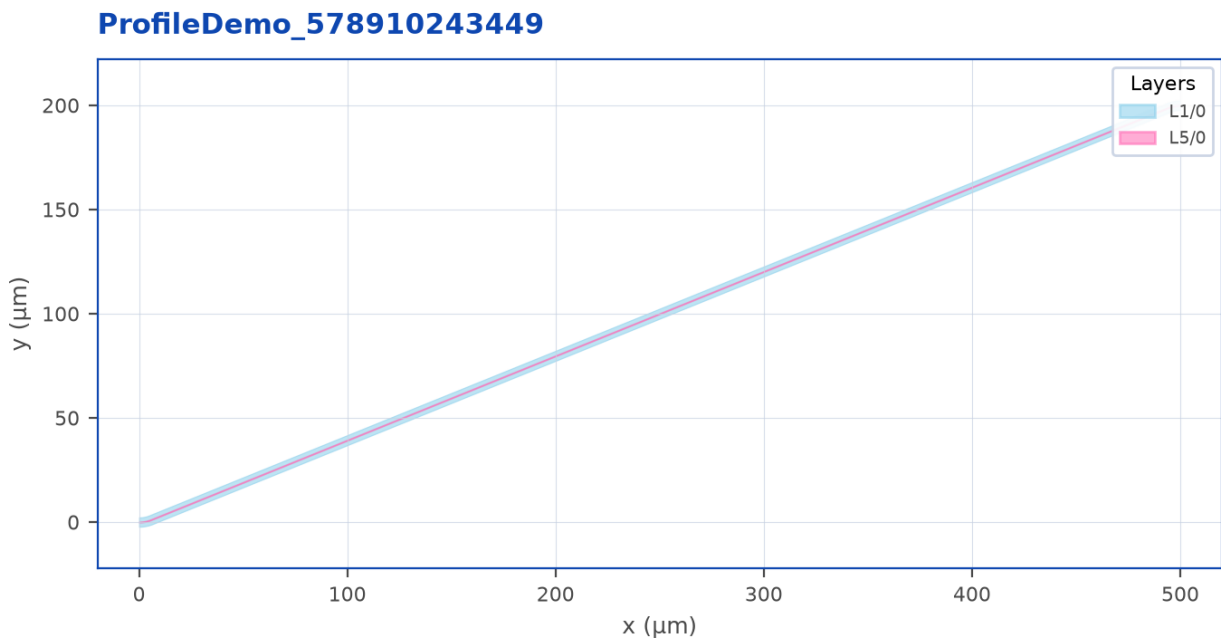


Figure 5.1.: Output of 12_route_profiles.py.

Three things happen automatically:

1. The route is stamped on the profile's primary layer (here, SiLN) at the profile's width.
2. The bend radius defaults to the profile's radius if you don't pass `radius=` on the `Route(...)` call.
3. Any sibling strokes the profile defines (claddings, slot rails) are emitted as parallel `FlexRoutes` alongside the primary.

💡 Tip

PDK components publish ports with a `route_profile` already attached. So `chip.Route(coupler.ports["o1"], modulator.ports["i1"])` - no kwargs at all - does the right

thing on a real PDK. This is why we recommend assigning profiles in the PDK rather than at the call site.

5.2 What a profile contains

A profile is built from two kinds of strokes:

- **Layer strokes:** continuous parallel bands at fixed widths and offsets. Used for the core, claddings, exclusion zones, slot-mode rails.
- **Tooth strokes:** periodic PCells laid along the route. Used for grating teeth, periodic deep-etch features, periodic electrical contacts, photonic-crystal slabs.

Together they let one profile describe geometry as exotic as “core + cladding + exclusion + grating teeth every 0.32 μm ” in a single `Route()` call.

5.3 When to override at the call site

Two override knobs on `Route(...)`:

- `profile=...` overrides any port-attached profile for this one route.
- `radius=...` overrides the layout radius. Width changes use the chain’s `.width(...)` or `.style(width=...)` methods; width is not a `Route` constructor argument.

```
# Use an explicit layout radius compatible with the PDK curvature rule.
chip.Route(a, b, radius=15)

# Pin a different profile for this one route.
chip.Route(a, b, profile=pdk.RP.silc_rib)
```

For *section*-level overrides inside a single `Route`, see `.style(profile=...)` in the previous chapter.

5.4 Auto layer transitions

When `port_a.layer != port_b.layer`, the router consults a **transition registry**. If the layer pair has a registered transition PCell (e.g. a `PhotonicElevator` for SiLC-to-SiLN), the router inserts it automatically and continues on the new layer.

13_layer_transition.py

```

"""Auto layer transitions – cross-layer routes, no manual PCells.

When port_a is on one layer and port_b is on another, the router checks
the registered TransitionRegistry. If a transition PCell exists for the
pair (e.g. `PhotonicElevator` for SiLC ↔ SiLN), the router drops it in
automatically and continues on the new layer.

`transition_offset` controls where along the route the transition sits.
"""

import lumicron as lm
import lumicron.pdks.elyon_demo.all as pdk

@lm.pcell
def LayerTransition():
    c = lm.CELL("LayerTransition")

    # Input on silicon, output on nitride.
    c.add(lm.PORT(
        "in", position=(0, 0), direction=0,
        route_profile=pdk.RP.silc_strip,
    ))
    c.add(lm.PORT(
        "out", position=(600, 0), direction=180,
        route_profile=pdk.RP.siln_strip,
    ))

    # The router consults pdk's default transition registry for the
    # SiLC → SiLN pair and inserts the elevator at offset=300 μm.
    c.Route(c.ports["in"], transition_offset=300).to(c.ports["out"])
    return c

if __name__ == "__main__":
    LayerTransition().to_gds("layer_transition.gds")

```

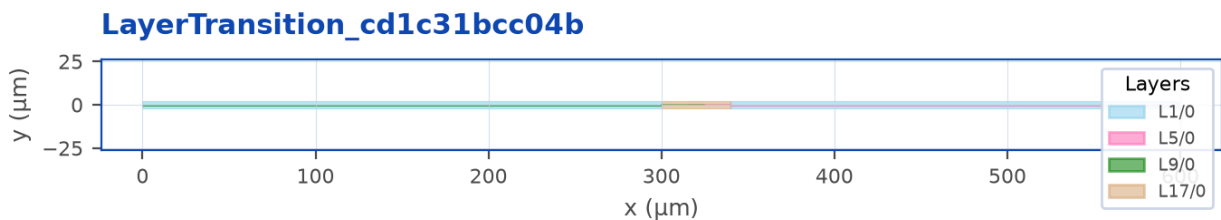


Figure 5.2.: Output of 13_layer_transition.py.

`transition_offset=300` places the transition PCell 300 μm in from port_a. PDK imports populate the default transition registry, so this works with no extra setup. To override, pass `transitions=` with a custom `TransitionRegistry`.

i Note

Section-level transitions (`.style(profile=...)` mid-chain) trigger the same registry. Whether you change layer at the start, in the middle, or at the end of a route, the same transition PCell drops in.

5.5 Where profiles come from

Every profile in this guide is built into the `elyon_demo` PDK. To define your own - including custom strokes, periodic teeth, and slot-mode rails - see the **PDK Authoring Guide**, which has a dedicated chapter on the `@route_profile` builder.

For now, treat profiles as a black box: pick one from your PDK's RP namespace, assign it to ports, and routes will inherit it.

CHAPTER 6

Bookmarks

A **bookmark** is a named bbox attached to a cell. In the Lumicron viewer it shows up in the Bookmarks panel; tapping it fits-to-screen on the geometry. Machine-readable test intent belongs to Registry, not Bookmarks.

Bookmarks restore a saved layout point/region/view, including camera state and optional notes. They are application metadata and never fabricated GDS/OASIS polygons. The application carries them in portable .lum annotation metadata.

6.1 Two ways to declare a bookmark

14_bookmarks.py

```
"""Bookmarks – labeled fit-to-screen markers for the viewer.

A bookmark is a named bbox attached to a cell. In the Lumicron viewer it
appears in the Bookmarks panel, and tapping it fits-to-screen on the
component. Two flavors:

    1. Free-floating: `c.add(lm.Bookmark("name", at=(x, y)))`
    2. Attached to a placement chain: `.add_bookmark(name="...")`
"""
import lumicron as lm
import lumicron.pdks.elyon_demo.all as pdk

@lm.pcell
def BookmarkDemo():
    c = lm.CELL("BookmarkDemo")

    pad = lm.Rectangle(x_dim=80, y_dim=80, layer=pdk.LAYER.MTL1)
```

```

# Bookmark attached to a placement – auto-fits the placed pad in the viewer.
c.add(pad)
c.Place(pad).at((0, 0)).add_bookmark(name="pad_1")

c.add(pad)
c.Place(pad).at((300, 0)).add_bookmark(name="pad_2", group="pads")

# Free-floating bookmark – bookmarks an arbitrary location with notes.
c.add(lm.Bookmark(
    "alignment_mark",
    at=(150, 150),
    notes="Coarse-align target – visible at 5× under microscope.",
))
return c

if __name__ == "__main__":
    BookmarkDemo().to_gds("bookmark_demo.gds")

```

1. **Attached to a placement chain** (`.add_bookmark(name=...)`) - the bookmark's bbox tracks the placement's world bbox after every `.at()`, `.move_by()`, `.rotate_by()`. This is the form to reach for almost always.
2. **Free-floating** (`c.add(lm.Bookmark("name", at=(x, y)))`) - for arbitrary annotations like alignment marks or zoom-to-region targets.

6.2 Groups

Pass `group="..."` to bind multiple bookmarks into a `BookmarkGroup`. The viewer shows groups as collapsible sections in the Bookmarks panel:

```

for i, position in enumerate(modulator_xs):
    chip.add(modulator)
    chip.Place(modulator).at((position, 0)) \
        .add_bookmark(name=f"mod_{i}", group="modulators")

```

All bookmarks sharing the same `group=` end up in a single `BookmarkGroup` when the application resolves the cell.

6.3 Notes

`notes=` accepts a markdown string. The viewer renders it in the device-info card when the bookmark is selected.

```
chip.add(lm.Bookmark(
    "alignment_mark",
    at=(150, 150),
    notes="Coarse-align target - visible at 5× under microscope.",
))
```

Notes are editable in-app too - a script can establish a default and the test engineer can refine it later without touching the script.

6.4 Auto-named bookmarks on placements

If you pass `bookmark=True` to `Place(...)` without a `bookmark_name=`, the bookmark is named after the placed cell:

```
chip.Place(mzi_cell, bookmark=True).at((0, 0)) # bookmark name: "MZI"
```

Pass `bookmark_name="..."` to override. This shorthand is equivalent to:

```
chip.Place(mzi_cell).at((0, 0)).add_bookmark(name="...")
```

6.5 Empty names are rejected

`lm.Bookmark("", at=(0, 0))` raises `ValueError`. Same for whitespace-only names and `add_bookmark(name="")`. This catches a frequent typo where a name slot is meant to receive a value from a config dict but the dict is missing the key.

6.6 Notebook Notes

A `Note` carries user-authored information. A `Bookmark` means “take me here”; a `Note` means “here is information about this design.” Keep them separate:

```
chip.add(lm.Note(id="review-1", title="Review coupling gap",
    body="Check this value before tapeout."))
```

Add `Notes` to the selected top cell. Child `Notes` are not automatically promoted. `targetBookmarkID` may reference a resolved `Bookmark` UUID when that identity is available; invalid targets are rejected at annotation export. A `Note` does not claim that a simulation or test ran or passed.

Legacy .lum annotation schema 1 and saved model Pin keys remain readable. New annotation output uses schema 2, bookmarks, authoredBookmarkIndex, and targetBookmarkID. Conflicting old/new keys fail explicitly.

CHAPTER 7

Registry and physical test manifests

Registry connects design intent to a physical test manifest. It does not run a laser, probe station or measurement recipe.

Two operations define the workflow:

- `lm.register(cell, ...)` marks an exact cell definition as a testable DUT.
- `lm.build_registry(root)` finds its registered **placed occurrences** anywhere inside that root's hierarchy.

An unregistered container does not stop discovery. If both a module and its children are registered, they are separate entries. The supplied root itself is a definition, not a placed occurrence, and is excluded. Building a DOE's Registry therefore gives its local DUTs; building TOP's Registry includes those DUTs through every placement of the DOE. No forwarding or publication step is required.

7.1 Declare named test ports

```
import lumicron as lm

core = lm.LayerSpec(layer=1)
dut = lm.CELL("LOSS_DUT")
guide = dut.add(lm.Waveguide([(0, 0), (200, 0)], width=0.5, layer=core))
guide.promote({"i1": "input", "o1": "output"})
lm.register(dut, ports=["input", "output"],
            data={"test_type": "spectral", "recipe": "cutback", "notes": "Reference
↪ first"})

doe = lm.CELL("LOSS_DOE")
first = doe.add(dut)
second = doe.add(dut)
```

```

doe.Place(second).at((0, 100))
top = lm.CELL("TOP")
block = top.add(doe)
top.Place(block).at((1000, 500))

local = lm.build_registry(doe) # two occurrences, DOE coordinates
registry = lm.build_registry(top) # two occurrences, TOP coordinates
print(registry.status)
for issue in registry.issues:
    print(issue.instance_path, issue.message)
registry.write_csv("test-manifest.csv")
registry.write_json("test-manifest.json")
top.to_gds("top.gds")

```

ports accepts a nonempty list or tuple of unique, nonempty strings. Names are exact and belong to the registered DUT's own declared interface. Missing ports make the entry and Registry **incomplete**; the DUT remains visible with an issue. Registry never searches descendants for an equal name. Use normal explicit port promotion when the DUT must expose a deeper interface.

Each port record contains its name, transformed position, outward direction, width, GDS layer/datatype, port type and route-profile name when present. Lengths are micrometres. Directions are degrees counter-clockwise from +X. Positions, directions and widths include every ancestor translation, rotation, reflection and magnification in the supplied root's frame. Port records are distinct from the DUT's reference origin and transform.

7.2 Existing physical I/O sites

```

lm.register(dut, io=GratingCoupler, data={"recipe": "fiber_alignment"})

```

The established `io=Factory` form selects exact `@pcell/@gds cell` factories below a DUT. These records retain the physical I/O **instance origins**, which may be fiber landing sites. They are not aliases for named PORT centers. The search remains recursive; a nested registered DUT owns its own I/O subtree and its sites are not also attributed to the outer DUT. Both registered DUTs still appear in the Registry. No matching site produces an explicit incomplete entry.

Supply exactly one of `io` or `ports`. Supplying both is an error. Re-registering the same definition intentionally replaces its declaration. A registration is shared by placements of that exact definition; an occurrence-specific metadata override is not provided in this phase.

7.3 Identity, hierarchy and validity

Entries use the existing authored instance path, not coordinates or a sibling slot, for DUT identity. Repeated placements remain separate. Moving a supported authored occurrence changes coordinates without transferring its identity. Hierarchy display names are context, not an identity substitute.

The existing provenance limits still apply: ambiguous repeated source assignments and arbitrary raw Python executions cannot establish regeneration continuity. Use unambiguous authored handles in the supported Script execution context, or persist an existing authored model. Registry does not invent identity matching.

ARRAY is geometry-only. Registry does not manufacture independently testable array occurrences. An array containing registered definitions produces an unsupported-target issue; use real placements for independently testable DUTs. Old schema-1/2 snapshots containing array records remain readable as historical unvalidated projections, not newly qualified test targets.

status is complete or incomplete for schema 3. A complete empty Registry is different from one whose child definition is missing. Structural issues identify missing definitions, unavailable selected interfaces and unsupported test targets. Malformed declarations, invalid data, duplicate authored identities and hierarchy cycles can reject the build outright. Neither path reports a valid partial result. Registry validation is not DRC, LVS or fabrication qualification.

data remains an opaque, deeply copied JSON dictionary. Strings, finite numbers, booleans, null, lists and string-keyed dictionaries are supported. Recipes are identifiers, not executable instructions. `CELL.add_info()` is independent; its values are never copied automatically into the manifest.

7.4 Native Registry Viewer

Generate a semantic layout after `build_registry(top)`, then use **Registry** in the Layout document bar. Rows retain hierarchy paths, validity, interface counts and physical locations. The **Within** filter narrows the already resolved Registry to a container; coordinates remain in the explicitly displayed Registry root frame. To obtain DOE-local coordinates, build/open the DOE's own layout.

DUT overlays outline the entire loaded DUT bounding box, transformed into the Registry root frame; they are display annotations, not mask geometry. Selecting a row centers that physical occurrence through the existing Layout navigation and shows a separate Registry summary in Inspector. Legacy I/O markers stay at their distinct site origins. Missing geometry does not acquire an invented box. Named test ports and opaque metadata are inspectable there. Reverse selection is available when the selected authored reference resolves to exactly one Registry occurrence;

ambiguous repeated definition-local selections are not guessed. A non-geometric issue does not acquire an invented marker.

Export writes CSV or JSON from the same resolved model. Incomplete exports remain visibly incomplete. Consumers must check status and issues. Old snapshots show **Legacy / unvalidated** rather than claiming modern validation.

7.5 Deterministic exports

Schema 3 JSON records the root, coordinateFrame: "root-cell", units: "um", status/issues and entries. An entry contains authored path, hierarchy, composed transform, selected/resolved named ports, separate legacy io sites and opaque data. Keys and interface/entry ordering are canonicalized. No timestamp or memory address is included. JSON/YAML snapshots are untested; only embedding in a validated semantic artifact binds a projection to exact generated geometry.

CSV has one row per resolved named port or I/O site, a DUT row when no interface resolved, and issue rows for failures without a resolved DUT. Column order is:

```
schema_version, root_cell, coordinate_frame, units, registry_status,
registration_id, authored_instance_path, instance_path, cell_name, status,
interface_kind, interface_id, port_name, x_um, y_um, direction_deg,
width_um, layer, datatype, route_profile, port_type, metadata_json, issues_json
```

Entries sort by ID; ports by name; sites by ID. Metadata/issues use escaped JSON cells rather than inventing team-specific columns. Unknown fields are blank. UTF-8 preserves Unicode. LF ends rows. Fields containing quotes, commas or newlines are quoted with doubled internal quotes. Numbers use each producer's locale-independent round-trip decimal representation; compare numeric values across Python and native exports, not their float formatting. Equivalent models produce stable bytes within each producer. CSV and JSON retain the same evidence.

7.6 Persistence and freshness

Authored source owns reproducible declarations; serialized internal Layout state retains declarations and authored IDs. GDS semantic sidecars and .lum retain resolved projections for inspection. A portable .lum is not a complete source project. Raw GDS import does not recreate registration intent. Imported cells can be explicitly registered after their supported ports are supplied.

Build again after edits. A new build uses current transforms and interfaces; deleted occurrences disappear, and removed selected ports create issues. Export recomputes an existing Registry rather than stamping old coordinates onto new geometry. Earlier returned Registry objects remain

snapshots. File replacement is staged after serialization; process-kill/power-loss durability is not claimed.

7.7 Complete qualification example

Examples/Phase10Registry contains MZI, ring, loss and legacy calibration DUTs, separate TOP/DOE builds, exports, failure/move fixtures and consume.py. The consumer uses only the Python standard library. It refuses incomplete/legacy inputs and produces a deterministic **hypothetical** queue with interfaces and metadata. No instrument is operated, and the fixture is not fabrication-qualified.

Part III.

Working with PDKs

CHAPTER 8

Working with PDKs

A **process design kit** packages everything a foundry’s process gives you:

- Named layers with design rules.
- Pre-made route profiles (`RP.silc_strip`, `RP.siln_strip`, ...).
- Components with authored geometry and declared ports (`EdgeCouplerSi`, `NxM_MMI`, `Photon-icElevator`, `modulators`, ...).
- A transition registry so cross-layer routes auto-insert the right PCell.

lumicon ships with `elyon_demo`, an internal PDK we use throughout this guide. Load foundry .lumpdk archives in Lumicon. For standalone Python use, install a corresponding Python distribution when the PDK publisher provides one; `pip` does not install .lumpdk archives.

8.1 Importing

```
import lumicon.pdks.elyon_demo.all as pdk
```

The `.all` re-export gives you the conventional namespaces:

Namespace	Contents
<code>pdk.LAYER</code>	Named layers (<code>SILC</code> , <code>SILN</code> , <code>MTL1</code> , ...).
<code>pdk.RP</code>	Route profiles (<code>silc_strip</code> , <code>siln_strip</code> , ...).
<code>pdk.<Component></code>	Components - <code>EdgeCouplerSi</code> , <code>NxM_MMI</code> , ...

The import line explicitly names the PDK the script targets. Keep it consistent with the PDK loaded into the active project. By convention, alias it to `pdk` as above.

8.2 Discovering what's in a PDK

LAYER and RP support `.print_rules()` for a quick reference card:

```
pdk.LAYER.print_rules()    # all layers + min_width / min_spacing / min_radius
pdk.RP.print_rules()      # all profiles + their cross-sections
```

Individual layers also support `.rules()` and inspecting fields directly (`layer.layer`, `layer.datatype`, `layer.color`, `layer.min_width`).

8.3 Using PDK components

PDK components are `@pcell`-decorated factories. Call them like functions; the returned cell is parametrized and auto-named:

```
coupler = pdk.EdgeCouplerSi(width=0.5)    # default-named cell
coupler_wide = pdk.EdgeCouplerSi(width=1.5) # different cell - different name
↪ auto-generated

a = c.add(coupler)
b = c.add(coupler_wide)
c.Place(a).at((0, 0))
c.Place(b).at((0, 1000))
```

The `@pcell` decorator derives cell identity from its factory, source, PDK, and effective typed parameters. Every call constructs a cell; it does not return a cached mutable object. To place two instances of one constructed definition, keep that cell and call `c.add(cell)` twice.

8.4 Component ports

When component ports carry compatible `route_profile` declarations, `c.Route(coupler.ports["o1"], modulator.ports["i1"])` can inherit routing defaults. Otherwise, provide a profile explicitly. Cross-layer routing requires a registered transition; the router does not infer an arbitrary physical transition from layer names.

Tip

This is the payoff for a well-curated PDK: scripts read like circuit diagrams. Once your team's PDK has profile-tagged ports on every component, end-user scripts shrink dramatically.

8.5 Loading a PDK in Lumicron

Use **File > Load PDK** or **Change PDK** in the project inspector to choose a `.lumpdk` archive. The active project's PDK metadata and Python package form the context for scripts, schematics, and layout inspection. Keep the script's explicit import consistent with that PDK.

A PDK has a human display name and a Python import identity `package-name` in `META`. Use the import name shown by the active PDK; do not derive Python names from display text. The retired `package_name` authoring key is rejected. The demo import above is for examples, not a replacement for a foundry PDK.

8.6 Writing your own PDK

Defining new layers, components, and route profiles for an in-house process is covered in the separate **PDK Authoring Guide**. That document includes a full chapter on the `@route_profile` builder - the API for declaring custom cross-sections with continuous strokes (`p.layer(...)`) and periodic features (`p.tooth(...)`).

CHAPTER 9

Packages and project environments

A reusable Lumicron package can be two files. The package defines engineering code; `lumicron`, `lumicron_pdk`, and `lumicron_validate` are the Python APIs that code uses.

Start with `Examples/Packages/SimplePDK` and `OBandLibrary` in the Lumicron repository. The complete ecosystem is small enough to read before learning the resolver.

9.1 Inspect an environment in Lumicron

Click the project/environment name at the bottom left of the window, or open **Script** → **Packages** in the inspector. The **Inspect** menu switches between **Project Environment** and each resolved package. Choosing a package changes the inspection view; it does not activate a different process or alter the layout.

The environment view lists packages, versions and capabilities. **Edit Dependencies...** opens the project's `lumicron.toml`; **Refresh Dependencies** checks Git branch tips and local packages. It asks before changing package versions or locked commits. Package Details shows the local source or Git locator, the locked Git commit where applicable, dependencies, and **Show Package in Finder**.

Technology and component packages expose Layers, Stack, PCells and Routes. A component library shows its own PCells and identifies the technology supplying its layers, stack and route profiles. Select O-band and C-band separately in the example to inspect their distinct Guide definitions. Validation packages list declared checks, not completed verification results. Utility packages list their entry files; inspect those files for their Python functions.

The environment is project-scoped. There is currently no app-wide package library or package browser. **Add Git Package...** adds a dependency from a repository URL; **Add Local Package...** lets you choose a package folder. Local packages remain in their source folders; Git packages are cached under the project's `.lumicron/` directory. Opening the project verifies an existing

inspection cache without importing code; use Refresh Dependencies when the cache is absent or local files have changed.

9.2 Start a project with local and Git dependencies

1. Use **New Project** in Lumicron. It creates an editable `lumicron.toml` and script.
2. In **Script** → **Packages** → **Project Environment**, choose **Add Git Package...**. Enter the repository URL, branch/tag/commit (HEAD uses the default branch), and optional package folder. Choose **Review** to fetch and read package metadata.
3. Review the declared package IDs, versions, capabilities, technology targets and exact Git commits, including new transitive dependencies. **Add to Project** runs the package definitions, saves the dependency and lock, and opens its inspector. Existing dependencies stay pinned; this action never silently replaces an existing package ID or advances other branches.
4. To add a local library, choose **Add Local Package...**, select its folder containing `lumicron.toml`, review the package IDs and capabilities, then choose **Add to Project**. Lumicron records a relative path from the project to that folder.
5. Select packages in the inspector to review the resolved resources.
6. In the design script, import the package by its Python name, for example `import example_oband.all` as `oband`. The native Run action activates the project's environment for that execution.

Review does not execute package Python or change the project's manifest, lock or inspection data; it can populate the project Git cache. Add to Project executes trusted package code to build inspection data. If import or a handled file-write failure occurs, the previous project files are retained/restored. A process kill or power loss is not a three-file transaction: a mixed artifact pair is rejected by existing digest validation. Changed project/package inputs invalidate the review.

Use HTTPS or SSH repository URLs with your existing Git credential helper or SSH agent. An absolute local Git repository path also works. Lumicron does not store passwords/tokens in this form. Hosting-service sign-in, SSH setup and remote repository creation are separate from adding an engineering package.

For manual manifest edits, choose **Edit Dependencies...** and add one table for each direct dependency. Libraries can declare their own technology. Save the manifest, then choose **Refresh Dependencies**. It shows available package versions and Git commits before updating the manifest and lock. You can keep the current pins.

For example, keep the generated `[project]` section and add:

```
[dependencies."example.oband"]
path = "../0BandLibrary"

[dependencies."team.routing"]
git = "ssh://git.example.org/team/routing.git"
ref = "main"
```

The Git URL is illustrative; substitute a repository containing a package whose manifest declares `team.routing`. Each dependency key must match that package's ID. Local paths are relative to the declaring manifest, so the first entry expects an `0BandLibrary` folder beside the project. Keep both the manifest and resolved lock in source control.

TOML declares **which packages belong to the environment**. Import a declared package as `<package_id_with_dots_and_hyphens_replaced_by_underscores>.all`. For example, `cornerstone.soi220.oband` becomes `cornerstone_soi220_oband.all`. The import uses the project's exact lock and keeps process ownership and environment evidence. Arbitrary folders are not registered packages.

Your project's own Git repository is separate from its dependencies. The Script **Git** navigator can initialize a local repository, stage and commit files, and open **Repository Settings...** to add a remote, fetch, pull (fast-forward only), or push. Create the remote repository with your hosting provider and configure authentication before connecting it. Commit authored sources, `lumicron.toml` and `lumicron.lock`; keep `.lumicron/` caches out of source control. Adding a package does not publish your project.

9.3 A tiny process and its component library

`SimplePDK/lumicron.toml` declares the identity and the file to load:

```
schema = 1
[package]
id = "example.simple"
version = "1.0.0"
[capabilities]
technology = "technology.py"
```

`technology.py` contains an ordinary layer table:

```
import lumicron_pdk as lpd

LAYER = lpd.LayerTable(
    CORE=lpd.Layer(1, 0, stipple=lpd.Stipple.SPARSE_DOTS)
)
```

A component library declares its own identity and names the process it uses:

```
schema = 1
[package]
id = "example.oband"
version = "0.1.0"
process = "example.simple"
[capabilities]
components = "components.py"
[dependencies,"example.simple"]
path = "../SimplePDK"
version = "1.0.0"
```

In `components.py`, import the declared process and define components with the existing Layout API:

```
import lumicron as lm
import example_simple.all as tech

@lm.pcell(symbol="waveguide")
def Guide(length=30):
    c = lm.CELL("Guide")
    c.Place(c.add(lm.Rectangle(length, 2, layer=tech.LAYER.CORE))).at((length / 2, 0))
    c.add(lm.PORT("o1", position=(0, 0), direction=180, width=2,
    ↪ layer=tech.LAYER.CORE))
    c.add(lm.PORT("o2", position=(length, 0), direction=0, width=2,
    ↪ layer=tech.LAYER.CORE))
    return c
```

No `__init__.py`, manual registry, or duplicate layer definition is required. The component's library is `example.oband`; its process is `example.simple`. The C-band example exposes another `Guide` without replacing this one.

9.4 A tapeout consumes packages

A project manifest has `[project]` instead of `[package]`. Its portable ID is created once and kept in source control. A clone keeps this semantic ID. Each checkout retains separate app-owned conversations and live verification sessions.

```
schema = 1
[project]
id = "my-team-tapeout"
[dependencies,"example.oband"]
path = "../OBandLibrary"
```

Run the project script in Lumicron. Its declared environment is activated for the run. From an ordinary Python process, activate it explicitly:

```
from lumicron.packages import resolve

environment = resolve("/path/to/tapeout")
with environment.activate():
    import example_oband.all as library
    component = library.Guide()
```

Package Python is trusted executable code. Resolving source bytes alone does not import it. Running a script, generating a native inspection projection, and qualification do execute package definitions. Activation is scoped and does not permit concurrent package environments in different Python threads. Native script runs use separate processes.

9.5 Requirements, lock, and local edits

`lumicron.toml` is authored intent. `lumicron.lock` records exact package IDs, versions, capabilities, process targets, dependencies, source locators, content SHA-256 digests, and Git revisions. Commit both files. Ignore `.lumicron/`, virtual environments, and Python caches.

`resolve()` creates a lock when none exists. With an existing lock it verifies requirements and package bytes. A changed local dependency is an error until the user chooses **Refresh Dependencies** in Lumicron. That action also checks Git branch tips and asks before changing package versions or locked commits. For an explicit terminal update:

```
python -m lumicron.packages resolve . --update
```

The command also executes package introspection and writes a derived native inspection cache. `python -m lumicron.packages inspect .` prints the verified lock without importing package code. Project opening reads the cache and checks its inputs; missing or changed inputs make the projection unavailable. It never updates a branch tip merely because a project was reopened.

Only exact version constraints are supported in this first resolver. One package ID has one resolved version. Conflicting sources/versions, cycles, missing dependencies, and undeclared dependency imports during package initialization fail visibly. Local paths are relative to the manifest that declares them. Package content includes ordinary source/assets, excluding Git metadata, `.lumicron`, Python caches and its root lock. Put generated outputs outside package source trees so they do not unexpectedly change content identity.

9.6 Git sources and offline reconstruction

Git is optional. Local-path projects work without it. A Git dependency uses standard Git transport, SSH, credential helpers and system configuration:

```
[dependencies."example.oband"]
git = "ssh://git.example.org/team/oband.git"
ref = "main"
# subdir = "packages/oband" # optional path inside that repository
```

The lock stores the resolved immutable commit. Updating the remote does not update a locked project. `resolve . --update` fetches intentionally. The exact cached revision works while the remote is unavailable. A clone without that cache needs the remote to reconstruct it. Relative dependencies of a Git package must remain inside its pinned repository. No GitHub login or provider integration is required or implied. Bundled source locators are reserved for shipped engineering package resources; this release ships the examples as ordinary local sources. A future registry can provide another source adapter; no registry is implemented.

9.7 Multiple processes, components and validation

The demonstration tapeout combines two process packages and two optical component libraries. A process owns its layer definitions. Libraries targeting the same process do not create duplicate layer owners. Native Layers groups include the process name; component catalogs retain qualified identities while displaying friendly names.

GDS has a global layer/datatype encoding. If two resolved processes declare the same pair, physical export from that environment fails, even if a particular layout does not use both layers. Lumicron does not remap fabrication layers or choose whichever technology loaded first. Use distinct process encodings or separate process outputs. Environment identity does not turn incompatible fabrication technologies into one process.

A validation package has a `validation` capability pointing to ordinary Python. The `SimpleValidation` example calls `lumicron_validate.dfx.port_orientation` with an authored instance ID and port. Utility packages work the same way: `GratingTools` exports a small alignment function and needs no process. Capability names are extensible; native views understand technology, components and validation.

Validation discovery lists provider IDs, process targets and explicit functions or DRC decks. Users call/select the provider deliberately. Decks are not merged. Native DRC/LVS and simulation retain their existing evidence scopes; this package model does not certify a multi-process native simulation or automatically bind a package provider to the native DRC session.

9.8 Organize your repository freely

New Project creates a manifest, a small `design/top.py`, `README` and `.gitignore`. These are editable starting files. Rename `design`, move the script, or put it at the root. Environment resolution does not inventory or prescribe design folders. `FlexibleTapeout` demonstrates an intentionally unusual structure.

```
python -m lumicron.packages new-project MyTapeout
python -m lumicron.packages initialize existing-repository
python -m lumicron.packages new-package MyComponents --id team.components --capability
↳ components
```

Initialization adds only minimal project metadata and preserves existing files. New Package and Initialize are terminal/backend operations in this phase; there is no separate native wizard. Technology, components, validation and utilities are templates over the same manifest, not different package systems.

9.9 Identity and generated evidence

Package ID answers who owns a definition. Process ID identifies its technology. Environment ID binds exact resolved package content and dependency revisions. Source locator answers where the bytes came from. Changing a remote URL does not rename a package. Absolute cache paths are excluded from environment identity.

Authored `lai_/lra_` identity remains separate. Another Git commit does not rename an authored object. Existing source-anchor continuity limits still apply when programs or source files change. Package/environment evidence is stored in reserved cell custom-data projections and travels with supported Layout/GDS semantic/.lum serialization. Digest association is local evidence, not cryptographic authenticity. A generated artifact can describe an older environment without being a current result for today's inputs.

9.9.1 Legacy archive compatibility

A resolved environment with no technology packages can still use the existing Load PDK `.lumpdk` workflow. Its archive digest remains the legacy evidence owner. Once a technology package is declared, the environment owns the process context; an unrelated archive cannot silently add another process. An unresolved or stale environment must be resolved before Lumicron can prove that this fallback applies. Use **Refresh Dependencies** in Script's Packages inspector (also available when inspection is unavailable), or the explicit terminal command above. New Project resolves its empty environment as part of creation.

9.10 Project branches, conflicts, and recovery

Git owns the repository, branch, index, commits and conflicts. Lumicron observes those records, including changes made in Terminal. Repository Settings supports branch changes, fetch, fast-forward pull and ordinary non-force push. Save dirty editors and preserve tracked changes before changing branches. Lumicron never silently stashes, discards changes or resolves a conflict. Detached HEAD remains a detached checkout; no branch is invented.

A branch can contain a different manifest and lock. Its old environment inspection cache then becomes unavailable. Choose **Refresh Dependencies** to reconstruct that branch's exact packages. An upstream update is a separate reviewed choice; **Keep Current** preserves the lock. An exact cached Git package remains usable offline even if its transport mirror was removed. Explicit Refresh can reconstruct a damaged owned checkout only when the reconstructed bytes agree with the lock. An unavailable remote still prevents fetching an uncached revision. Mutable local packages need their authored files restored or an explicit refresh of their new content; they are never replaced as if they were disposable Git cache.

Resolve conflict markers in `lumicron.toml` before resolving dependency intent; resolve markers in `lumicron.lock` before trusting exact resolution. Conflicted Schematic and Notebook documents retain their bytes and require conflict resolution. Lumicron does not choose either side or offer semantic merging. After repairing the files, refresh the project and its dependencies.

Commit authored sources, package declarations and locks. Generated engineering artifacts and historical results can be ordinary versioned files when your team chooses to retain them. `.lumicron/`, interpreter caches and virtual environments are machine-local. Existing projects' ignore rules are not automatically rewritten. A project copy preserves its semantic identity; app-owned sessions remain separate. Relative local dependencies must move with their required relative arrangement. Historical run snapshots retain their original package/model inputs and paths; a new Git HEAD does not rewrite them or prove them current.

9.11 Run the project pipeline locally

A pipeline is one ordinary project-owned command. Lumicron runs the command; your script, test tool or Makefile owns its checks and any step names. Add this optional table to the existing `schema1` project manifest:

```
[pipeline]
command = ["/bin/sh", "pipeline.sh"]
```

For example, `pipeline.sh` can invoke the same checks used by CI:

```
#!/bin/sh
set -eu
: "${LUMICRON_PYTHON:?Set the supported Lumicron Python interpreter}"
exec "$LUMICRON_PYTHON" -B checks.py
```

The Script **Git** navigator's **Run Pipeline Locally...** opens the command and output view. Running requires an explicit button press and saved editors. Opening or inspecting a project never executes its pipeline. The working directory is the project root. `LUMICRON_PYTHON` supplies the selected interpreter and `LUMICRON_PROJECT_ROOT` supplies the root path. Python checks use the same existing `packages.resolve(root).activate()` context when importing engineering packages. There is no separate pipeline implementation of Layout, DRC, DFX, Registry or LVS.

A terminal or CI worker using the supported runtime can run the identical binding:

```
python -m lumicron.packages._pipeline run /path/to/project
```

This private application adapter requires an existing exact lock and validates package inputs before running. It does not update dependencies. Resolve first when setting up a fresh clone. It also rejects changed manifest, lock or package inputs after execution. Script arguments are passed literally; shell expansion occurs only when the configured command explicitly invokes a shell.

The view shows live output, actual exit status, duration and cancellation. Logs are retained under `.lumicron/pipeline-runs/`. Native runs have a 30-minute limit and retain up to 2 MiB stdout plus 64 KiB stderr, with explicit truncation notice. The result describes that invocation, not a permanent current-design certificate. Fix a failing check or configuration and run again. A project command is executable content: review it before running it. Lumicron adds no commit, stage, push, reset, clean or checkout; if your command performs one, the refreshed Git state shows it. Hosted CI YAML interpretation, cloud dashboards and provider authentication are outside this local-command boundary.

When a fresh clone has an exact lock but no available inspection projection, **Keep Current** reconstructs that locked environment after declining an update. It does not rewrite the manifest or lock, accept changed local package bytes, or advance the reviewed Git revision. An already available projection is left alone.

Part IV.

Cookbook

CHAPTER 10

Cookbook

Three end-to-end recipes that combine everything from the previous chapters. Each one is a standalone script you can run as `python <name>.py`.

10.1 Mach-Zehnder interferometer

An MZI: Y-splitter, two arms of different geometry, Y-combiner. The top arm carries a chained taper pair (a phase-shifter region); the bottom arm jogs south for a delay.

`15_mzi.py`

```
"""Cookbook: a Mach-Zehnder interferometer.

Two paths between a Y-splitter and a Y-combiner. The top arm carries a
chained taper pair (a thermal phase shifter region); the bottom arm
takes a delay via .jog(). This example does not request length matching.
"""

import lumicon as lm
import lumicon.pdks.elyon_demo.all as pdk

@lm.pcell
def YSplitter(width: float = 0.5):
    """Stand-in Y-junction with three ports: i1, o1, o2."""
    c = lm.CELL("YSplitter")
    body = lm.Rectangle(x_dim=20, y_dim=4, layer=pdk.LAYER.SILC)
    c.add(body)
    c.Place(body).at((0, 0))
    c.add(lm.PORT("i1", position=(0, 0), direction=180,
                  width=width, route_profile=pdk.RP.silc_strip))
    c.add(lm.PORT("o1", position=(20, 1.5), direction=0,
                  width=width, route_profile=pdk.RP.silc_strip))
    c.add(lm.PORT("o2", position=(20, -1.5), direction=0,
```

```

        width=width, route_profile=pdk.RP.silc_strip))

    return c

@lm.pcell
def MZI(arm_length: float = 400, delay: float = 60):
    c = lm.CELL("MZI")

    s = c.add(YSplitter())
    k = c.add(YSplitter())

    c.Place(s).at((0, 0))
    c.Place(k).at((arm_length + 60, 0)).rotate_by(180)

    # Top arm – straight, with a phase-shifter taper bay.
    (c.Route(s.ports["o1"], radius=15, bend="euler").to(k.ports["o2"])
     .taper(offset=120, width=2.0, length=20)
     .taper(offset=120, width=0.5, length=20))

    # Bottom arm – jog south by `delay` to add a delay region.
    c.Route(s.ports["o2"], radius=15, bend="euler").to(k.ports["o1"]) \
        .jog("S", by=delay)
    return c

if __name__ == "__main__":
    MZI().to_gds("mzi.gds")

```

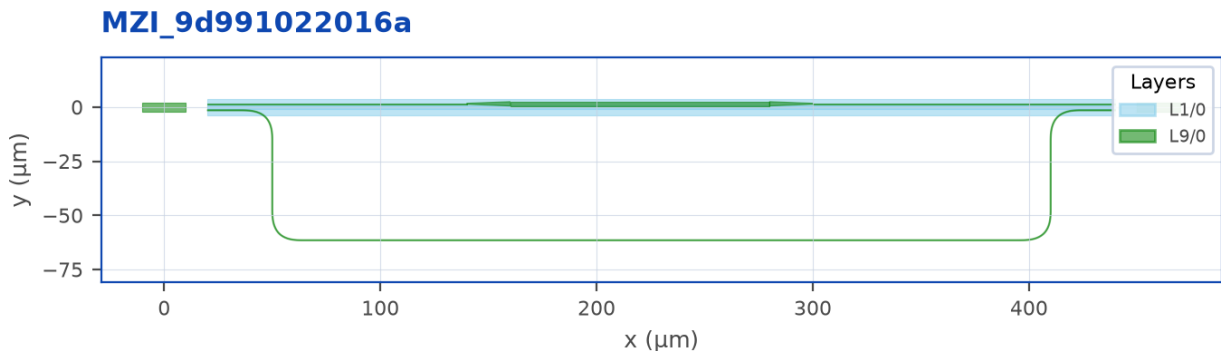


Figure 10.1.: Output of 15_mzi.py.

What to notice:

- `@lm.pcell` turns `YSplitter` into a parametrized cell factory. Two calls with the same parameters share the same cell definition under the hood; only the references differ.
- **Combiner is rotated 180°** so its `o1` and `o2` face left toward the splitter's arm ports.
- `.taper(...)` is chained twice to widen, hold, and narrow the top arm. Width carries forward

between chained tapers.

- `.jog("S", by=delay)` on the bottom arm is the entire delay region - no manual segments.

This example does not request length matching or establish equal arm lengths. For an explicit ΔL , retain the route handles and use the length-matching workflow in the Routing chapter.

10.2 Ring resonator

A single-bus, single-ring add/drop. The ring is just a `lm.Ring` shape; its position is determined by `radius + width/2 + gap` from the bus center.

16_ring_resonator.py

```

"""Cookbook: a single-bus ring resonator.

A bus waveguide running east-west, a ring sitting above it with the
gap controlled by `gap`. The ring is just a `lm.Ring` shape on the same
layer as the bus.
"""

import lumicron as lm
import lumicron.pdks.elyon_demo.all as pdk

@lm.pcell
def RingResonator(
    radius: float = 25,
    width: float = 0.5,
    gap: float = 0.2,
    bus_length: float = 200,
):
    c = lm.CELL("RingResonator")

    # Bus waveguide: a flat rectangle along the silicon strip layer.
    bus = lm.Rectangle(x_dim=bus_length, y_dim=width, layer=pdk.LAYER.SILC)
    c.add(bus)
    c.Place(bus).at((0, 0))

    # Ring centered above the bus, separated by gap.
    ring_y = width / 2 + gap + radius
    ring = lm.Ring(outer_radius=radius, width=width, layer=pdk.LAYER.SILC)
    c.add(ring)
    c.Place(ring).at((bus_length / 2, ring_y))

    c.add(lm.PORT("in", position=(0, 0), direction=180,
                  width=width, layer=pdk.LAYER.SILC))
    c.add(lm.PORT("out", position=(bus_length, 0), direction=0,
                  width=width, layer=pdk.LAYER.SILC))

    return c

```

```
if __name__ == "__main__":
    RingResonator().to_gds("ring_resonator.gds")
```

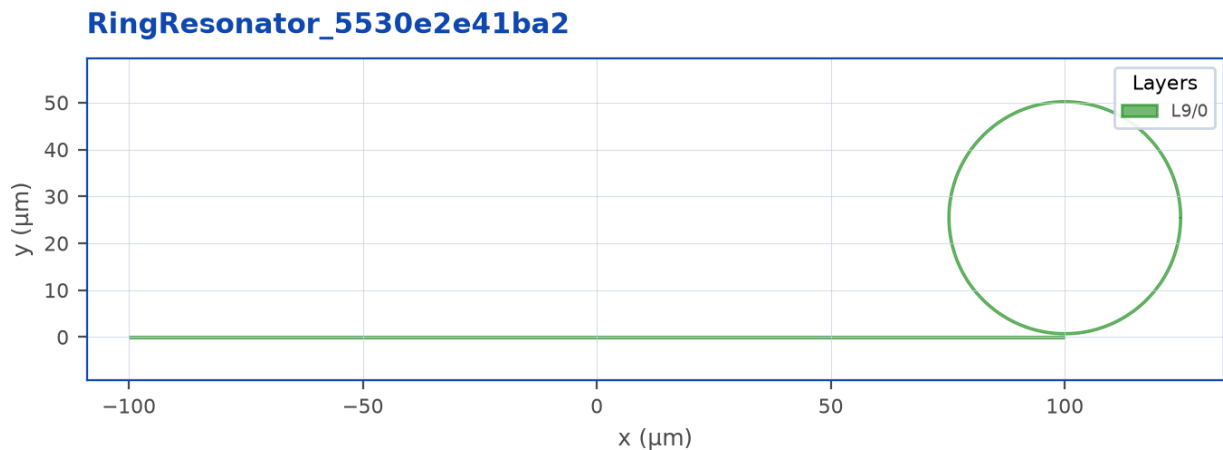


Figure 10.2.: Output of 16_ring_resonator.py.

For an add/drop (two-bus) ring, add a second Rectangle on the opposite side of the ring at the same gap and a second pair of ports. To sweep radius or coupling gap across a die, wrap this in `@lm.pcell(radius=..., gap=...)` and place the array.

10.3 Fanout bus

Four channels entering at 10 μm pitch, leaving at 50 μm. The bundle absorbs the pitch difference with auto-fanouts - no manual U-detours.

17_fanout_bus.py

```
"""Cookbook: a tight bundle fanned out to a wider port pitch.

A 4-channel bundle entering at 10 μm pitch and leaving at 50 μm. The
auto-fanout absorbs the pitch difference into a U-shape on each side;
the spine in between stays parallel and tight.
"""

import lumicon as lm
import lumicon.pdks.elyon_demo.all as pdk

@lm.pcell
def FanoutBus(
    n: int = 4,
```

```

pitch_in: float = 10.0,
pitch_out: float = 50.0,
span: float = 600.0,
):
    c = lm.CELL("FanoutBus")

    for i in range(n):
        c.add(lm.PORT(
            f"in{i}", position=(0, i * pitch_in), direction=0,
            route_profile=pdk.RP.silc_strip,
        ))
        c.add(lm.PORT(
            f"out{i}", position=(span, i * pitch_out), direction=180,
            route_profile=pdk.RP.silc_strip,
        ))

    a = [c.ports[f"in{i}"] for i in range(n)]
    b = [c.ports[f"out{i}"] for i in range(n)]
    c.Route(a, spacing=pitch_in, radius=15).to(b)
    return c

if __name__ == "__main__":
    FanoutBus().to_gds("fanout_bus.gds")

```

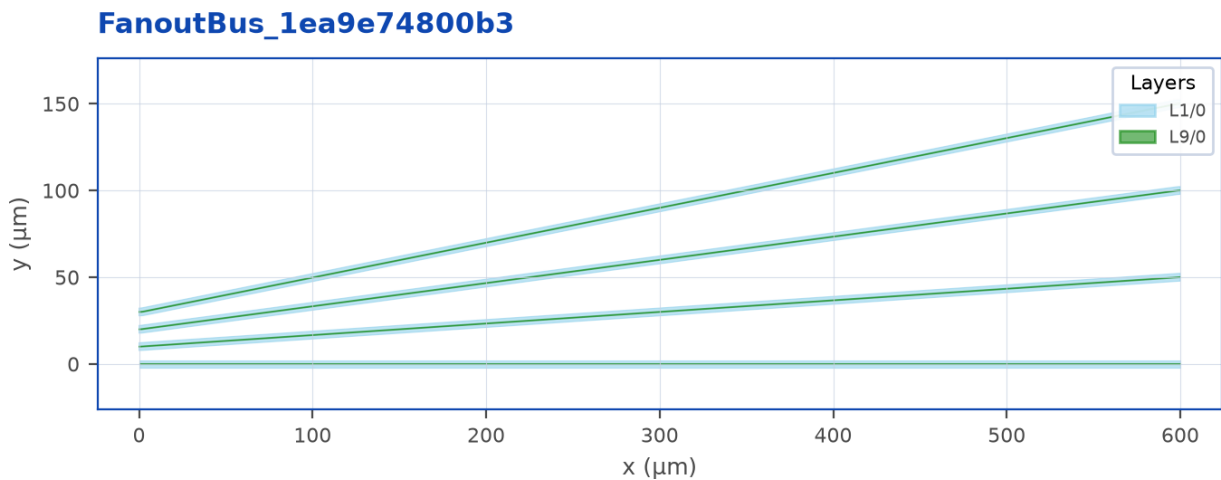


Figure 10.3.: Output of 17_fanout_bus.py.

Why this matters: when you connect a 10 μm -pitched modulator array to a 127 μm -pitched fiber array, the channel-by-channel fanout you'd write by hand is identical for every chip you ever build. Route does it once, correctly, with bend-radius-aware staircase math.

For tighter pitches where the staircase math wastes space, pass `fanout="sbend"` for a curved fanout instead.

Part V.

Agent Workflows

CHAPTER 11

Working with Phoebe

Phoebe uses the same Lumicron API reference as this guide. Documentation describes API behavior; current project facts come from project inspection tools. Neither source substitutes for the other.

11.1 Reference questions

Ask about an API symbol or a behavior, for example `Join`, endpoint straight lengths, route profiles, or simulation bench inputs. Phoebe can retrieve relevant sections locally. API lookup does not execute scripts, change the project, or require a commercial connection. An answer should identify the canonical spelling and the reference section used.

11.2 Project questions

To explain why a particular route or placement differs from your intent, Phoebe needs the actual source and relevant layout or schematic facts. Enable the relevant context in Assistant Settings. A documentation example cannot establish which ports, parameters, or model your project actually uses. Missing metadata should be reported as unavailable, not guessed.

11.3 Actions and reasoning

Apple's on-device model handles supported simple questions. More involved tasks may use the installed local model or a connected commercial service through Phoebe's routing. Local reference lookup remains available to all of those backends. Source edits and execution follow Lumicron's action permissions and review workflow; a proposed action is not a completed action.

For a diagnosis, include the observed result and intended behavior, let Phoebe inspect the relevant evidence, review any proposed change, and check the regenerated layout. A generated layout is not fabrication approval or proof that a physical model matches the foundry device.

CHAPTER A

Cheat sheet

A compact reference. Skim this once before reading anything else.

A.1 Boilerplate

```
import lumicron as lm
import lumicron.pdks.elyon_demo.all as pdk

chip = lm.CELL("Chip")
# ... build ...
chip.to_gds("chip.gds")
```

A.2 Cells & shapes

Call	What you get
lm.CELL("Name")	New cell
lm.Rectangle(x_dim=, y_dim=, layer=)	Centered rectangle
lm.Rectangle(..., centered=False)	Origin at lower-left
lm.Circle(radius=, layer=)	Filled circle
lm.Ring(outer_radius=, width=, layer=)	Annulus
lm.Polygon(points=[(x,y), ...], layer=)	Arbitrary polygon
lm.ARRAY(child, count=, pitch=(dx, dy))	Periodic array
c.add(thing)	Add to cell, returns handle

A.3 Placement

Chain method	Effect
<code>.at((x, y))</code>	Set the selected anchor's target
<code>.using("W"/"E"/"N"/"S"/"NE"/...)</code>	Use a child anchor instead of its origin
<code>.using("C")</code> (not <code>.using(handle.C)</code>)	Select a local anchor; handle properties are world points
<code>.rotate_by(deg)</code>	Rotate CCW
<code>.move_by(dx, dy)</code>	Accumulate a parent-coordinate shift for references
<code>.right_of(other), .left_of(other)</code>	Anchor-to-anchor align
<code>.above(other), .below(other)</code>	Anchor-to-anchor align
<code>.using("i1").at(other.ports["o1"])</code>	Snap a port onto another (translate only)

Call	Effect
<code>c.Join(a.ports["in"], b.ports["out"])</code>	Mate two ports: a's cell moves + rotates onto b's port; warns on width/layer mismatch

A.4 Ports

Call	What it does
<code>lm.PORT("name", position=, direction=, width=, layer=)</code>	Define a port
<code>lm.PORT(..., port_type="electrical")</code>	Electrical (vs optical)
<code>lm.PORT(..., route_profile=...)</code>	Attach a profile
<code>c.ports["name"] / handle.ports["name"]</code>	Look up a port
<code>handle.promote(ports=[...], prefix="...")</code>	Re-publish child ports
<code>c.show_ports()</code>	Print port table

Direction: degrees CCW from +X. 0 = E, 90 = N, 180 = W, 270 = S.

A.5 Routing - global kwargs

```
chip.Route(a, b,
           radius=10,           # bend radius (μm)
           bend="circular",     # "circular" | "euler" | CELL
           p=0.5,              # euler fraction (0-1)
           profile=pdk.RP.silc_strip, # cross-section recipe
           style=None,         # automatic; or "manhattan", "direct", "sbend"
           start_straight=0,    # min straight after port_a
           end_straight=0,      # min straight before port_b
           clearance=0,         # detour around obstacles by this μm
           transition_offset=50, # auto layer-transition position
           transitions=None)    # custom TransitionRegistry
```

A.6 Routing - chain methods

Method	Effect
<code>.go("E"/"N"/"W"/"S", by=, to=)</code>	Manhattan section
<code>.go("NE"/"NW"/"SE"/"SW", by=(dx, dy))</code>	S-bend section
<code>.jog(dir, by=)</code>	U-shaped detour
<code>.style(arg=, profile=, layer=, radius=, width=, bend=)</code>	Section-level style overrides
<code>.style("default")</code>	Revert all section overrides
<code>.radius(r)</code>	Override radius for next section
<code>.bend(b, p=)</code>	Override bend type for next section
<code>.width(w)</code>	Override width - auto-taper on boundary
<code>.taper(offset=, width=, length=)</code>	Insert a positional taper
<code>.start_taper(width=, length=)</code>	Configure port_a-side taper
<code>.end_taper(width=, length=)</code>	Configure port_b-side taper
<code>.match_length_to(other, delta=, ...)</code>	Length-match this route to another

A.7 Routing bundles

```
bundle = chip.Route(ports_a, ports_b,
                   spacing=10, radius=15,
                   style="manhattan", sort=False)
bundle.match_lengths() # target, delta, mirror, tolerance
first_channel = bundle.routes[0]
```

Then call `.go(...)` for Manhattan-mode bundles, or chain `.taper(...)` / `.match_lengths(...)` to broadcast.

A.8 Bookmarks

```
chip.Place(thing).at(...).add_bookmark(name="...", group="...", notes="...")
chip.add(lm.Bookmark("alignment", at=(x, y)))
chip.add(lm.Bookmark("dicing_lane", bbox=(xmin, ymin, xmax, ymax)))
```

A.9 Output

```
chip.to_gds("chip.gds")
chip.to_oas("chip.oas")      # OASIS
chip.to_gds(max_points=199)  # legacy GDS-II compat
```

A.10 Useful introspection

```
c.show_ports()
pdk.LAYER.print_rules()
pdk.RP.print_rules()
any_obj.rules()      # most objects support this
```

CHAPTER B

Errors & warnings

A reference for the messages you'll see while authoring scripts. All errors are subclasses of `Exception`; all warnings are subclasses of `UserWarning`.

B.1 Errors

B.1.1 `RouteError`

The auto-router or a chain method couldn't proceed. The message names the concrete cause; the most common variants:

Message contains	Likely cause	Fix
"no entry in cell.styles"	<code>.style("name")</code> referenced a name not in the cell's style table.	Add <code>c.styles["name"] = RouteProfile(...)</code> first, or pass the object directly.
"S-bend forward must be positive"	<code>.go("NW", ...)</code> from an east-facing port - the diagonal would go backwards.	Pick a diagonal that advances in the port's direction.
"no transition registered for layer pair"	Cross-layer route with no transition PCell in the registry.	Pass <code>transitions=</code> with a registry that handles the pair, or stay on one layer.
"off-angle ports require flatten"	Cell hierarchy has off-cardinal ports at a junction.	Add <code>flatten=True</code> on the offending cell, or simplify hierarchy.

B.1.2 GeometryError

A shape's parameters are inconsistent (e.g. negative dimensions, inverted bbox).

B.1.3 ValueError from `lm.Bookmark(...)` / `add_bookmark(...)`

- "requires a non-empty name" - empty / whitespace-only name.
- "requires either at= or bbox=" - neither given.
- "bbox corners are inverted" - $x_{\max} < x_{\min}$ or $y_{\max} < y_{\min}$.

B.1.4 TypeError

You passed the wrong kind of object to a chain method. Most common: `chip.Place(some_handle)` where `some_handle` was already a `PlacementChain` (call `.Place` on the original cell, not on a chain return value).

B.1.5 ValueError from `CELL.Join(...)`

The moving port must come from a placed handle in the current cell. The fixed port must come from another handle in that cell or from the current cell's own ports mapping. A raw cell-definition port has no placement to move; one `ArrayElementHandle` cannot move independently of its shared array transform; and a consumed or foreign handle is rejected.

B.2 Warnings

Warnings don't stop your script. They surface in the terminal and (when running inside the Lumicron app) in the Output panel.

B.2.1 GeometryWarning

Message contains	Cause	Action
"S-bend radius ... below minimum"	The radius you passed is too tight for the route's forward / lateral budget.	Increase <code>radius=</code> , lengthen the forward span, or reduce the lateral offset.

Message contains	Cause	Action
“radius ... below layer min_radius”	radius= is below the route layer’s min_radius rule.	Increase radius=, or change to a layer with a tighter rule.
“bridge taper auto-inserted at port_b”	Final width didn’t equal port_b.width.	Add an explicit .end_taper(...) or finish with .width(port_b.width).

B.2.2 RouteWarning

Message contains	Cause
“match_length_to: target shorter than current”	The route is already longer than the match target - no meander added.
“amplitude= miss”	A pinned amplitude= produced a length that misses the target by > 1 μm.

B.2.3 ProfileWarning

Message contains	Cause
“radius=... overrides the profile’s radius”	You passed radius= <i>and</i> a profile carrying its own. The kwarg wins.
“tooth strokes unsupported with custom bend PCell”	A profile’s tooth segments can’t follow a PCell’s curve. Bend PCells render their own geometry.

B.2.4 Join warnings

CELL.Join emits ordinary Python warnings when the two ports disagree on width, layer, port type, or named route profile. The joint is still made because explicit cross-section changes can be intentional. Verify the junction geometry or insert a real transition when the mismatch is not deliberate.

B.3 Suppressing warnings

Standard Python `warnings.filterwarnings` works:

```
import warnings
import lumicron as lm
warnings.filterwarnings("ignore", category=lm.GeometryWarning)
```

But warnings exist for a reason - usually a silent geometry compromise. Prefer fixing the cause.

CHAPTER C

Glossary

Terms used throughout this guide.

AREF - Array reference. A GDS construct that places a child cell on a regular grid as a single record, instead of N independent SRefs. `lm.ARRAY` emits one AREF per call.

Bend - How a route makes a turn. "circular" is a constant-radius arc; "euler" is a clothoid where curvature ramps linearly from zero at entry, peaks mid-bend, and ramps back to zero at exit. Custom bends are 90° PCells.

Bundle - A group of parallel routes. Created with `c.Route(...)`.

Cell - A reusable hierarchical container of geometry, ports, sub-cells, and bookmarks. `lm.CELL("Name")` creates one.

Datatype - The second number in a (layer, datatype) pair. Foundries often use it to subclassify shapes on the same layer (e.g. layer 9 datatype 0 = drawn silicon, datatype 1 = exclusion).

Direct routing - A route geometry that uses Dubins arcs (arc-straight-arc) instead of Manhattan elbows. Picked automatically for non-cardinal port pairs.

Dubins path - The shortest curve connecting two oriented points subject to a turning radius constraint. Lumicon's direct router uses Dubins paths to route between non-cardinal ports.

Fanout - The geometry that absorbs a port-pitch mismatch at the entry / exit of a bundle. Manhattan fanout uses staircased U-detours; S-bend fanout uses a 4-point diagonal bridge.

FlexRoute / FlexPath - gdstk's parameterized polyline-with-bends primitive. Lumicon's auto-router emits FlexRoute records for most paths.

Handle - The object returned by `c.add(...)`. For sub-cells and arrays it carries placement and port-access methods.

Layer - A (layer_number, datatype) pair that names a process step (silicon, metal-1, oxide, ...). PDK layers are Layer objects with design rules attached.

Manhattan routing - A route geometry where every segment is axis-aligned and turns are 90° elbows.

OASIS - A GDS-II successor format, more compact for large layouts. `to_gds("foo.oas")` writes OASIS.

PCell - A parametric cell. In Lumicron, any function decorated with `@lm.pcell` is a PCell - its parameters are hashed into the cell name so distinct calls produce distinct GDS cells.

PDK - Process Design Kit. A package of layers, design rules, components, route profiles, and transitions for a specific foundry process.

Bookmark - A named bbox bookmark on a cell. Surfaces in the Lumicron viewer's Bookmarks panel. Not a GDS label.

Port - A named, oriented connection point on a cell. Carries a position, direction, width, layer, and optional route profile.

BookmarkGroup - A collection of bookmarks sharing a `group="..."` tag. Surfaces as a collapsible section in the viewer.

Route - A connection between two ports, emitted as one or more FlexRoute records plus optional taper / transition / bend PCells.

Route profile - A cross-section recipe: layers, widths, periodic features. Built with the `@route_profile` decorator (covered in the *PDK Authoring Guide*).

S-bend - A two-curve transition that shifts a route laterally without changing its primary direction. `c.Route(a, b).go("NE", by=(dx, dy))` lays one.

SRef - Single Reference. A GDS placement of a child cell. `c.add(child).Place(...).at(...)` emits an SRef.

Section (of a route) - A stretch of a `Route()` chain between two `.go()` / `.jog()` calls (or between the start / a `.go()`, or a `.go()` / the end). Section-level chain modifiers (`.style()`, `.radius()`, `.bend()`, `.width()`) apply to the *next* section.

Transition - A PCell that bridges two layers (e.g. SiLC-to-SiLN). Registered in a `TransitionRegistry`; the router consults it when a route crosses layers.

API reference

Signatures below are generated from the bundled Lumicron API. Use the task-oriented chapters for examples and constraints. Builder class names describe return values; users normally obtain them through CELL actions and chained methods.

Im.AND

Intersection: $a \cap b$ — keeps only the overlap.

```
lm.AND(a, b, *, layer: 'LayerSpec | None' = None, precision: 'float' = 0.001, name:  
↪ 'str | None' = None)
```

Im.ARRAY

Array specification — creates a 1-D or 2-D array of cell instances.

```
lm.ARRAY(component, count: 'int | tuple[int, int]', pitch: 'tuple[float, float]',  
↪ params: 'dict | None' = None, rotation: 'float' = 0)
```

Im.Arc

Arc segment — a curved band between two angles.

```
lm.Arc(radius: 'float', width: 'float', start_angle: 'float', end_angle: 'float',  
↪ layer: 'LayerSpec', num_points: 'int | None' = None, center: 'tuple[float, float]'  
↪ = (0.0, 0.0)) -> 'Polygon'
```

Im.Bookmark

User-facing factory for free-floating bookmarks. Pass to `c.add()`.

```
Im.Bookmark(name: 'str', *, at: 'tuple[float, float] | None' = None, bbox:
↳ 'tuple[float, float, float, float] | None' = None, group: 'str | None' = None,
↳ notes: 'str | None' = None) -> 'None'
```

Im.CELL

Cell builder — the primary object users interact with.

```
Im.CELL(name: 'str | None' = None)
```

Im.CellFromGDS

Import one GDS cell as an ordinary, self-contained Lumicron CELL.

```
Im.CellFromGDS(path: 'str | Path', top_cell: 'str | None' = None) -> 'CELL'
```

Im.Circle

Circle approximated as a regular polygon.

```
Im.Circle(radius: 'float', layer: 'LayerSpec', num_points: 'int | None' = None,
↳ center: 'tuple[float, float]' = (0.0, 0.0)) -> 'Polygon'
```

Im.LayerSpec

A GDS-II layer specified by layer number and datatype number.

```
Im.LayerSpec(*, layer: Annotated[int, Ge(ge=0), Le(le=65535)], datatype:
↳ Annotated[int, Ge(ge=0), Le(le=65535)] = 0) -> None
```

Im.NOT

Difference: $a - b$ — subtracts b from a .

```
Im.NOT(a, b, *, layer: 'LayerSpec | None' = None, precision: 'float' = 0.001, name:
↳ 'str | None' = None)
```

Im.Note

A user-authored note for explicit import into the app's Notebook.

```
Im.Note(*, id: Annotated[str, MinLen(min_length=1), MaxLen(max_length=128),
↳ _PydanticGeneralMetadata(pattern='^[A-Za-z0-9_.-]+$', title: Annotated[str,
↳ MinLen(min_length=1), MaxLen(max_length=4096)]], body: Annotated[str,
↳ MaxLen(max_length=65536)] = '', targetBookmarkID: uuid.UUID | None = None) -> None
```

Im.OR

Union / merge: $a \cup b$.

```
Im.OR(a, b=None, *, layer: 'LayerSpec | None' = None, precision: 'float' = 0.001,
↳ name: 'str | None' = None)
```

Im.Oval

Ellipse (oval) approximated as a polygon.

```
Im.Oval(rx: 'float', ry: 'float', layer: 'LayerSpec', num_points: 'int | None' = None,
↳ center: 'tuple[float, float]' = (0.0, 0.0), rotation_deg: 'float' = 0.0) ->
↳ 'Polygon'
```

Im.PORT

Port specification for adding ports to a cell.

```
lm.PORT(name: 'str', position: 'tuple[float, float]' = (0.0, 0.0), direction: 'float'
↳ = 0.0, width: 'float | None' = None, layer: 'LayerSpec | None' = None,
↳ route_profile: 'RouteProfile | None' = None, port_type: 'str' = 'optical')
```

lm.Parallelogram

Quadrilateral with independent top/bottom widths and x offsets.

```
lm.Parallelogram(top_width: 'float', bot_width: 'float', y_dim: 'float', layer:
↳ 'LayerSpec', top_x_offset: 'float' = 0, bot_x_offset: 'float' = 0, centered:
↳ 'bool' = True) -> 'Polygon'
```

lm.Polygon

Arbitrary closed polygon defined by a list of vertices.

```
lm.Polygon(*, type: Literal['polygon'] = 'polygon', vertices:
↳ Annotated[list[tuple[float, float]], MinLen(min_length=3)], holes:
↳ list[list[tuple[float, float]]] = <factory>, layer:
↳ lumicron._technology.layer.LayerSpec, source_location:
↳ lumicron.models.elements.SourceLoc | None = None, authored_id: str | None =
↳ <factory>, semantic_route_group: str | None = None, semantic_route_endpoints:
↳ list[dict] | None = None, semantic_route_name: str | None = None,
↳ semantic_route_kind: str | None = None, semantic_route_port_type:
↳ Literal['optical', 'electrical'] = 'optical', semantic_route_role: str | None =
↳ None, semantic_profile_name: str | None = None, semantic_propagation_model: dict |
↳ None = None, semantic_stroke_index: Annotated[int | None, Ge(gt=0)] = None,
↳ semantic_centerline: list[tuple[float, float]] | None = None, semantic_width:
↳ Annotated[float | None, Gt(gt=0)] = None, semantic_width_end: Annotated[float |
↳ None, Gt(gt=0)] = None) -> None
```

lm.Rectangle

Axis-aligned rectangle.

```
lm.Rectangle(x_dim: 'float', y_dim: 'float', layer: 'LayerSpec', centered: 'bool' =
↳ True) -> 'Rect'
```

Im.Ring

Ring (annulus) — outer circle minus inner circle.

```
lm.Ring(outer_radius: 'float', width: 'float', layer: 'LayerSpec', num_points: 'int |
↳ None' = None, center: 'tuple[float, float]' = (0.0, 0.0)) -> 'Polygon'
```

Im.Trapezoid

Symmetric trapezoid.

```
lm.Trapezoid(top_width: 'float', bottom_width: 'float', height: 'float', layer:
↳ 'LayerSpec', centered: 'bool' = True) -> 'Polygon'
```

Im.Triangle

Isosceles triangle.

```
lm.Triangle(base: 'float', height: 'float', layer: 'LayerSpec', centered: 'bool' =
↳ True) -> 'Polygon'
```

Im.Waveguide

Create a waveguide cell from a list of centerline points.

```
lm.Waveguide(points: 'list[tuple[float, float]]', layer: 'LayerSpec | None' = None,
↳ width: 'float | None' = None, width_end: 'float | None' = None, radius: 'float |
↳ None' = None, bend=None, p: 'float | None' = None, profile: 'RouteProfile | None' =
↳ None, name: 'str | None' = None, port_type: 'str' = 'optical')
```

Im.build_registry

Resolve every placed registered DUT beneath top.

```
lm.build_registry(top: 'CELL | Layout') -> 'DeviceRegistry'
```

lm.gds cell

Decorate an executable GDS-backed factory, with PCell parameters.

```
lm.gds cell(fn=None, *, symbol: 'str | None' = None, schematic: 'bool' = True)
```

lm.pcell

Decorator that marks a function as a parametric cell (PCell).

```
lm.pcell(fn=None, *, symbol: 'str | None' = None, schematic: 'bool' = True)
```

lm.register

Register one exact cell definition as a device under test.

```
lm.register(cell: '_CellT', *, io: 'PCell | None' = None, ports: 'list[str] |  
↳ tuple[str, ...] | None' = None, data: 'dict[str, Any] | None' = None) -> '_CellT'
```

lm.packages.PackageError

A package graph cannot be resolved without guessing or stale inputs.

lm.packages.Environment

Resolved engineering packages, independent from Python ABI and app session.

```
lm.packages.Environment(root, manifest, lock, records)
```

lm.packages.resolve

Resolve once or reconstruct the existing exact lock.

```
lm.packages.resolve(directory, *, update=False) ->  
↳ lumicron.packages._runtime.Environment
```

lm.packages.initialize

Add project metadata to an existing directory without moving its files.

```
lm.packages.initialize(directory)
```

lm.packages.create_project

Create an editable tapeout template; all authored directories may move.

```
lm.packages.create_project(directory)
```

lm.packages.create_package

Create a manifest and one capability entry file, with no empty scaffolding.

```
lm.packages.create_package(directory, *, identity, capability)
```

JoinChain

A selected moving port; to(target) mates it and returns its handle.

```
JoinChain(cell, source)
```

RouteRequest

A source port (or bundle) and route options, awaiting to(target).

```
RouteRequest(cell, source, source_location, options)
```

PlacementChain

Fluent placement API.

```
PlacementChain(handle: 'PlacedHandle', resolver=None, cell_builder=None)
```

RouteChain

Fluent routing API.

```
RouteChain(cell_builder: 'CELL', port_a: 'Port', port_b: 'Port', radius: 'float |
↳ None' = None, bend=None, p: 'float | None' = None, profile: 'RouteProfile | None' =
↳ None, style: 'str | None' = None, clearance: 'float' = 0.0, start_straight:
↳ 'float' = 0.0, end_straight: 'float' = 0.0, fit: 'bool' = False, source_loc=None,
↳ name: 'str | None' = None, semantic_group: 'str | None' = None)
```

RouteBundleChain

Private bundle machinery returned by `c.Route(port_list, port_list)`.

```
RouteBundleChain(cell_builder, ports_a, ports_b, routes: 'list[RouteChain]', spacing:
↳ 'float', radius: 'float', bend, p, profile, style, clearance: 'float',
↳ transition_offset: 'float', transitions, fanout: 'str' = 'U', target_length=None,
↳ start_straight: 'float' = 0.0, end_straight: 'float' = 0.0)
```

RefHandle

Handle for a single cell reference (SRef).

```
RefHandle(component: 'CELL', cellref)
```

ArrayHandle

Handle for an array of cell references (ARef).

```
ArrayHandle(component: 'CELL', cellref, columns: 'int', rows: 'int', pitch:
↳ 'tuple[float, float]', is_2d: 'bool' = False)
```

ArrayElementHandle

Handle for one element of an array (arr[col, row] or arr[i]).

```
ArrayElementHandle(array: 'ArrayHandle', col: 'int', row: 'int')
```

Im.CELL.C

Public API symbol exported by this build.

```
Im.CELL.C # property
```

Im.CELL.E

Public API symbol exported by this build.

```
Im.CELL.E # property
```

Im.CELL.Fill

Describe passive fill: Fill(tile).within(area).excluding(devices).

```
Im.CELL.Fill(tile)
```

Im.CELL.Join

Mate ports with c.Join(a).to(b); the two-port form remains supported.

```
Im.CELL.Join(port_a: 'Port', port_b: 'Port | None' = None) -> 'PlacedHandle |  
↔ JoinChain'
```

Im.CELL.N

Public API symbol exported by this build.

```
Im.CELL.N # property
```

Im.CELL.NE

Public API symbol exported by this build.

```
Im.CELL.NE # property
```

Im.CELL.NW

Public API symbol exported by this build.

```
Im.CELL.NW # property
```

Im.CELL.Place

Begin a placement chain for a previously added object or handle.

```
Im.CELL.Place(target, *, bookmark: 'bool' = False, bookmark_name: 'str | None' = None,
↳ bookmark_group: 'str | None' = None) -> 'PlacementChain'
```

Im.CELL.Route

Route with `c.Route(source, options...).to(destination)`.

```
Im.CELL.Route(port_a: 'Port | list[Port] | tuple[Port, ...]', port_b: 'Port |
↳ list[Port] | tuple[Port, ...] | None' = None, radius: 'float | None' = None,
↳ bend=None, p: 'float | None' = None, profile: 'RouteProfile | None' = None, style:
↳ 'str | None' = None, clearance: 'float' = 0.0, start_straight: 'float | None' =
↳ None, end_straight: 'float | None' = None, fit: 'bool' = False, transition_offset:
↳ 'float' = 50.0, transitions=None, target_length: 'float | None' = None, name: 'str
↳ | None' = None, *, spacing: 'float' = 10.0, fanout: 'str' = 'U', sort: 'bool' =
↳ False) -> 'RouteChain | RouteBundleChain | RouteRequest'
```

Im.CELL.S

Public API symbol exported by this build.

```
Im.CELL.S # property
```

Im.CELL.SE

Public API symbol exported by this build.

```
Im.CELL.SE # property
```

Im.CELL.SW

Public API symbol exported by this build.

```
Im.CELL.SW # property
```

Im.CELL.W

Public API symbol exported by this build.

```
Im.CELL.W # property
```

Im.CELL.add

Add an object to this cell.

```
Im.CELL.add(spec) -> 'PlacedHandle | None'
```

Im.CELL.add_data

Merge JSON metadata into this cell; supplied keys replace old values.

```
lm.CELL.add_data(data: 'dict') -> 'CELL'
```

lm.CELL.add_info

Merge neutral user information into this cell definition and return it.

```
lm.CELL.add_info(data: 'dict') -> 'CELL'
```

lm.CELL.bbox

Physical bounds of a resolved, independent geometry snapshot.

```
lm.CELL.bbox # property
```

lm.CELL.cell

The underlying data-model Cell.

```
lm.CELL.cell # property
```

lm.CELL.extract_netlist

Extract the existing physical netlist from a resolved snapshot.

```
lm.CELL.extract_netlist(*, extraction_tolerance: 'float' = 0.001,  
↪ transition_tolerance: 'float' = 0.0)
```

lm.CELL.get_data

Return copied metadata, or one value (a missing key raises KeyError).

```
lm.CELL.get_data(key: 'str | None' = None)
```

Im.CELL.inspect

Inspect a resolved snapshot as readable text or a structured dictionary.

```
Im.CELL.inspect(*, as_dict: 'bool' = False) -> 'str | dict'
```

Im.CELL.name

Public API symbol exported by this build.

```
Im.CELL.name # property
```

Im.CELL.ports

Public accessor for ports on this cell, indexed by name.

```
Im.CELL.ports # property
```

Im.CELL.rules

Print CELL API reference.

```
Im.CELL.rules()
```

Im.CELL.show_ports

Print a formatted summary of all ports on this cell.

```
Im.CELL.show_ports()
```

Im.CELL.to_gds

Resolve and write GDS, with the existing semantic sidecar contract.

```
lm.CELL.to_gds(path=None, *, flatten: 'bool' = False, max_points: 'int' = 0) -> 'None'
```

lm.CELL.to_oas

Resolve and write geometry-only OASIS; annotations are not polygons.

```
lm.CELL.to_oas(path=None, *, flatten: 'bool' = False) -> 'None'
```

lm.CELL.to_schematic

Persist a new Schematic from explicit child/Route/Join/PORT aliases.

```
lm.CELL.to_schematic(path=None)
```

lm.PORT.to_port

Public API symbol exported by this build.

```
lm.PORT.to_port() -> 'Port'
```

lm.packages.Environment.activate

Public API symbol exported by this build.

```
lm.packages.Environment.activate()
```

lm.packages.Environment.binding

Portable process/package evidence; never absolute cache locations.

```
lm.packages.Environment.binding()
```

lm.packages.Environment.load

Public API symbol exported by this build.

```
lm.packages.Environment.load(identity)
```

lm.packages.Environment.origin

Public API symbol exported by this build.

```
lm.packages.Environment.origin(identity)
```

lm.packages.Environment.verify

Reject changed package bytes before using a resolved snapshot.

```
lm.packages.Environment.verify()
```

lm.packages.Environment.write_projection

Publish a byte-bound native inspection cache after explicit execution.

```
lm.packages.Environment.write_projection()
```

JoinChain.to

Move the selected instance onto a fixed port in the same parent cell.

```
JoinChain.to(target: 'Port') -> 'PlacedHandle'
```

RouteRequest.to

Complete once, preserving scalar/bundle routing and its modifiers.

```
RouteRequest.to(target: 'Port | list[Port] | tuple[Port, ...]') -> 'RouteChain |  
↳ RouteBundleChain'
```

PlacementChain.above

Place this object above *other*, centered on other's x.

```
PlacementChain.above(other) -> 'PlacementChain'
```

PlacementChain.add_bookmark

Attach a bookmark to this placement.

```
PlacementChain.add_bookmark(name: 'str | None' = None, *, group: 'str | None' = None,  
↳ notes: 'str | None' = None) -> 'PlacementChain'
```

PlacementChain.aligned_x

Override the x-coordinate to match another anchor's x.

```
PlacementChain.aligned_x(point: 'tuple[float, float]') -> 'PlacementChain'
```

PlacementChain.aligned_y

Override the y-coordinate to match another anchor's y.

```
PlacementChain.aligned_y(point: 'tuple[float, float]') -> 'PlacementChain'
```

PlacementChain.at

Set the selected anchor's destination in parent-cell coordinates.

```
PlacementChain.at(position: 'tuple[float, float] | Port') -> 'PlacementChain'
```

PlacementChain.below

Place this object below *other*, centered on other's x.

```
PlacementChain.below(other) -> 'PlacementChain'
```

PlacementChain.flipX

Reflect in parent X (across Y), preserving the placement anchor.

```
PlacementChain.flipX() -> 'PlacementChain'
```

PlacementChain.flipY

Reflect in parent Y (across X), preserving the placement anchor.

```
PlacementChain.flipY() -> 'PlacementChain'
```

PlacementChain.left_of

Place this object to the left of *other*, centered on other's y.

```
PlacementChain.left_of(other) -> 'PlacementChain'
```

PlacementChain.move_by

Shift the placement by (dx, dy). Accumulates across calls.

```
PlacementChain.move_by(dx: 'float' = 0, dy: 'float' = 0) -> 'PlacementChain'
```

PlacementChain.relative_to

Set the reference point (anchor on another object).

```
PlacementChain.relative_to(point: 'tuple[float, float]') -> 'PlacementChain'
```

PlacementChain.right_of

Place this object to the right of *other*, centered on other's y.

```
PlacementChain.right_of(other) -> 'PlacementChain'
```

PlacementChain.rotate_by

Rotate by *angle* degrees CCW, then reapply the selected anchor/constraints.

```
PlacementChain.rotate_by(angle: 'float') -> 'PlacementChain'
```

PlacementChain.using

Select a semantic anchor, a resolved point, or independent X/Y selectors.

```
PlacementChain.using(anchor, y_selector=None) -> 'PlacementChain'
```

RouteChain.bend

Override the bend at subsequent sections.

```
RouteChain.bend(bend, p: 'float | None' = None) -> 'RouteChain'
```

RouteChain.end_taper

Configure the taper at the end (port B side) of the route.

```
RouteChain.end_taper(length: 'float | None' = None, cell: 'object | None' = None,  
↳ disabled: 'bool' = False, at: 'float | None' = None, width: 'float | None' = None)  
↳ -> 'RouteChain'
```

RouteChain.go

Add a manual routing segment.

```
RouteChain.go(direction: 'str', *, by=None, to=None, dx: 'float' = 0, dy: 'float' = 0)
↳ -> 'RouteChain'
```

RouteChain.jog

Shift the route by an offset, creating a proper manhattan detour.

```
RouteChain.jog(direction: 'str', by: 'float') -> "'RouteChain'"
```

RouteChain.length

Realized logical centerline length, including tapers, in microns.

```
RouteChain.length # property
```

RouteChain.match_length_to

Lengthen this route to match another route's path length + delta.

```
RouteChain.match_length_to(other: 'RouteChain', delta: 'float' = 0.0, at: 'float |
↳ None' = None, span: 'float | None' = None, amplitude: 'float | None' = None, side:
↳ 'str' = 'auto', tolerance: 'float' = 0.001) -> 'RouteChain'
```

RouteChain.name

Optional user-facing name shared by this route's emitted objects.

```
RouteChain.name # property
```

RouteChain.port_directions

Respect terminal directions, or ignore them for DC metal routing.

```
RouteChain.port_directions(policy: 'str') -> 'RouteChain'
```

RouteChain.radius

Override bend radius for subsequent sections.

```
RouteChain.radius(r: 'float | str') -> 'RouteChain'
```

RouteChain.rules

Print Route API reference.

```
RouteChain.rules()
```

RouteChain.start_taper

Configure the taper at the start (port A side) of the route.

```
RouteChain.start_taper(length: 'float | None' = None, cell: 'object | None' = None,
↳ disabled: 'bool' = False, at: 'float | None' = None, width: 'float | None' = None)
↳ -> 'RouteChain'
```

RouteChain.style

Configure one or more section-level overrides.

```
RouteChain.style(arg=None, *, profile: 'RouteProfile | str | None' = None, layer:
↳ 'LayerSpec | None' = None, radius: 'float | str | None' = None, width: 'float | str
↳ | None' = None, bend=None) -> 'RouteChain'
```

RouteChain.taper

Insert a positional taper into the route at chain order.

```
RouteChain.taper(offset: 'float' = 0.0, *, width: 'float | None' = None, length:  
↳ 'float | None' = None, cell: 'object | None' = None) -> 'RouteChain'
```

RouteChain.width

Override width for subsequent sections.

```
RouteChain.width(w: 'float | str') -> 'RouteChain'
```

RouteBundleChain.bend

Set the subsequent bend style on every channel.

```
RouteBundleChain.bend(value, p=None) -> "'RouteBundleChain'"
```

RouteBundleChain.finalize

Re-run the channel rebuild in strict mode, surfacing any bend-radius violations that were tolerated during intermediate `.go()` calls. Called automatically by `cell._to_layout()`.

```
RouteBundleChain.finalize() -> "'RouteBundleChain'"
```

RouteBundleChain.go

Drive the bundle's spine one manhattan leg.

```
RouteBundleChain.go(direction: 'str', *, by=None, to=None, dx: 'float' = 0, dy:  
↳ 'float' = 0) -> "'RouteBundleChain'"
```

RouteBundleChain.jog

Offset the spine sideways — a U-shaped detour, same idea as `RouteChain.jog` but applied to every channel through coordinated staircase corners. This manually steers the spine, so exact bundle length matching is unavailable afterward.

```
RouteBundleChain.jog(direction: 'str', by: 'float') -> "'RouteBundleChain'"
```

RouteBundleChain.match_length_to

Length-match this bundle to another route or bundle.

```
RouteBundleChain.match_length_to(other, delta=0.0, **kwargs) -> "'RouteBundleChain'"
```

RouteBundleChain.match_lengths

Equalize automatically routed channel lengths within this bundle.

```
RouteBundleChain.match_lengths(*, target: 'float | None' = None, delta: 'float |  
↪ list[float]' = 0.0, mirror: 'bool | list[bool]' = False, tolerance: 'float' =  
↪ 0.001) -> "'RouteBundleChain'"
```

RouteBundleChain.port_directions

Apply the electrical terminal-direction policy to all channels.

```
RouteBundleChain.port_directions(policy: 'str') -> "'RouteBundleChain'"
```

RouteBundleChain.radius

Set a shared bend radius before steering the bundle spine.

```
RouteBundleChain.radius(value) -> "'RouteBundleChain'"
```

RouteBundleChain.routes

Per-channel RouteChain objects, in ports_a order.

```
RouteBundleChain.routes # property
```

RouteBundleChain.style

Apply one section style/profile to every channel.

```
RouteBundleChain.style(*args, **kwargs) -> "'RouteBundleChain'"
```

RouteBundleChain.taper

Add a taper to each channel. Kwargs accept scalar or per-channel list.

```
RouteBundleChain.taper(**kwargs) -> "'RouteBundleChain'"
```

RouteBundleChain.width

Set the subsequent width on every channel (or a per-channel list).

```
RouteBundleChain.width(value) -> "'RouteBundleChain'"
```

RefHandle.flipX

Reflect in the parent X direction, keeping this handle's center fixed.

```
RefHandle.flipX() -> 'None'
```

RefHandle.flipY

Reflect in the parent Y direction, keeping this handle's center fixed.

```
RefHandle.flipY() -> 'None'
```

RefHandle.local_anchor

Public API symbol exported by this build.

```
RefHandle.local_anchor(key: 'str') -> 'tuple[float, float]'
```

RefHandle.local_bbox

Public API symbol exported by this build.

```
RefHandle.local_bbox # property
```

RefHandle.ports

Read-only mapping of ports in the placed coordinate frame.

```
RefHandle.ports # property
```

RefHandle.promote

Publish this placed cell's ports on the parent cell.

```
RefHandle.promote(ports=None, prefix: 'str' = '') -> 'RefHandle'
```

RefHandle.rotate

Rotate about the handle's world center by *angle* degrees CCW.

```
RefHandle.rotate(angle: 'float') -> 'None'
```

RefHandle.set_origin

Public API symbol exported by this build.

```
RefHandle.set_origin(x: 'float', y: 'float') -> 'None'
```

RefHandle.world_anchor

Public API symbol exported by this build.

```
RefHandle.world_anchor(key: 'str') -> 'tuple[float, float]'
```

RefHandle.world_bbox

Public API symbol exported by this build.

```
RefHandle.world_bbox # property
```

ArrayHandle.flipX

Reflect the array in parent X, preserving its center and repetition.

```
ArrayHandle.flipX() -> 'None'
```

ArrayHandle.flipY

Reflect the array in parent Y, preserving its center and repetition.

```
ArrayHandle.flipY() -> 'None'
```

ArrayHandle.local_anchor

Public API symbol exported by this build.

```
ArrayHandle.local_anchor(key: 'str') -> 'tuple[float, float]'
```

ArrayHandle.local_bbox

Public API symbol exported by this build.

```
ArrayHandle.local_bbox # property
```

ArrayHandle.rotate

Rotate the entire array as a group.

```
ArrayHandle.rotate(angle: 'float') -> 'None'
```

ArrayHandle.set_origin

Public API symbol exported by this build.

```
ArrayHandle.set_origin(x: 'float', y: 'float') -> 'None'
```

ArrayHandle.world_bbox

Public API symbol exported by this build.

```
ArrayHandle.world_bbox # property
```

ArrayElementHandle.C

Public API symbol exported by this build.

```
ArrayElementHandle.C # property
```

ArrayElementHandle.E

Public API symbol exported by this build.

```
ArrayElementHandle.E # property
```

ArrayElementHandle.N

Public API symbol exported by this build.

```
ArrayElementHandle.N # property
```

ArrayElementHandle.NE

Public API symbol exported by this build.

```
ArrayElementHandle.NE # property
```

ArrayElementHandle.NW

Public API symbol exported by this build.

```
ArrayElementHandle.NW # property
```

ArrayElementHandle.S

Public API symbol exported by this build.

```
ArrayElementHandle.S # property
```

ArrayElementHandle.SE

Public API symbol exported by this build.

```
ArrayElementHandle.SE # property
```

ArrayElementHandle.SW

Public API symbol exported by this build.

```
ArrayElementHandle.SW # property
```

ArrayElementHandle.W

Public API symbol exported by this build.

```
ArrayElementHandle.W # property
```

ArrayElementHandle.origin

This occurrence's transformed reference origin (read-only).

```
ArrayElementHandle.origin # property
```

ArrayElementHandle.world_anchor

Public API symbol exported by this build.

```
ArrayElementHandle.world_anchor(key: 'str') -> 'tuple[float, float]'
```

ArrayElementHandle.world_bbox

Public API symbol exported by this build.

```
ArrayElementHandle.world_bbox # property
```