

LUMICRON

PHOTONIC LAYOUT, FROM PYTHON.

User Guide

The `lumicron_api` Python package

Version	0.5.0
Released	May 2026
Publisher	Elyon Semiconductor

Lumicron API User Guide

© 2026 Elyon Semiconductor. All rights reserved.

Lumicron, the Lumicron wordmark, and `lumicron_api` are trademarks of Elyon Semiconductor.

This document is distributed alongside `lumicron_api` on PyPI and the Lumicron application bundle.

Code listings in this guide are extracted from the `user_docs/examples/` directory of the source tree and are tested at build time.

Photonic Layout, From Python.

Table of contents

Welcome	1
Preface	2
Who this is for	2
What's covered	2
What's <i>not</i> covered	2
Conventions	3
How to read	3
I. Getting Started	4
1. Getting started	5
1.1. Install	5
1.2. Hello, Lumicron	5
1.3. A more interesting first chip	7
1.4. What you just used	8
2. Core concepts	10
2.1. Naming convention	10
2.2. Layers	11
2.3. Cells	12
2.4. Ports	12
2.5. Layouts	13
II. Building Layouts	14
3. Placement	15
3.1. The chain	15
3.2. Anchors	16
3.3. Arrays	17
3.4. Hierarchy and re-use	17
3.5. Promoting child ports	18
4. Routing	20
4.1. Auto-routing	20

4.2. Bends	21
4.3. Manual routing with <code>.go()</code> and <code>.jog()</code>	22
4.4. RouteBundle	22
4.5. Length matching	24
4.5.1. Bundle PLM — rectangular detour	24
4.6. Section-level overrides	26
4.7. Tapers	27
4.8. Layout-aware routing	27
5. Route profiles	34
5.1. Using a profile	34
5.2. What a profile contains	36
5.3. When to override at the call site	36
5.4. Auto layer transitions	36
5.5. Where profiles come from	37
6. Pins	39
6.1. Two ways to declare a pin	39
6.2. Groups	39
6.3. Notes	40
6.4. Auto-named pins on placements	40
6.5. Empty names are rejected	40
III. Working with PDKs	42
7. Working with PDKs	43
7.1. Importing	43
7.2. Discovering what's in a PDK	44
7.3. Using PDK components	44
7.4. Component ports	44
7.5. Installing additional PDKs	45
7.6. Writing your own PDK	45
IV. Cookbook	46
8. Cookbook	47
8.1. Mach-Zehnder interferometer	47
8.2. Ring resonator	48
8.3. Fanout bus	48
Appendices	53
Cheat sheet	53
Boilerplate	53
Cells & shapes	53
Placement	53

Ports	54
Routing — global kwargs	54
Routing — chain methods	55
RouteBundle	55
Pins	55
Output	56
Useful introspection	56
Errors & warnings	57
Errors	57
RouteError	57
GeometryError	57
ValueError from <code>lm.PIN(...)</code> / <code>add_pin(...)</code>	58
TypeError	58
Warnings	58
GeometryWarning	58
RouteWarning	58
ProfileWarning	59
Suppressing warnings	59
Glossary	60

Welcome

This is the User Guide for **lumicron_api** — the Python package that lets you describe a photonic or electronic chip layout in code and emit a clean, foundry-ready GDS or OASIS file.

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

- **Cells** are reusable building blocks with named **ports**.
- **Place** instantiates cells; **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, where every script result renders live as you save.

This guide covers the standalone Python package. Authoring your own PDK is a separate document; the App/ viewer has its own.

Tip

Every code listing in this book is a real, executable file in `user_docs/examples/`. The build script runs them, regenerates figures, and rebuilds this PDF — so the snippets and the screenshots can never go out of sync.

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): install, the smallest possible script, the core nouns and verbs of the API.
- **Building layouts** (Chapters 3–6): placement, routing (auto and manual), route profiles, and pins.
- **Working with PDKs** (Chapter 7): using a process design kit's layers and components.
- **Cookbook** (Chapter 8): full recipes for an MZI, a ring resonator, and a fanout bus.
- **Appendices**: a one-page cheat sheet, an errors-and-warnings reference, and a glossary.

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 / contributing to `lumicron_api`** — see `developer_docs/` in the repository.

Conventions

Code blocks appear as:

```
import lumicron_api 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 Install

Lumicron requires Python 3.12 or newer.

```
pip install lumicron_api
```

That brings in the public API (`lumicron_api`), the `gdstk` GDS/OASIS backend, and the demo PDK (`lumicron_api.pdks.elyon_demo`). No system dependencies.

1.2 Hello, Lumicron

The smallest meaningful script: one cell, one rectangle, one GDS file.

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.

Listing 1.1 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_api 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_layout().to_gds("hello_lumicron.gds")
```

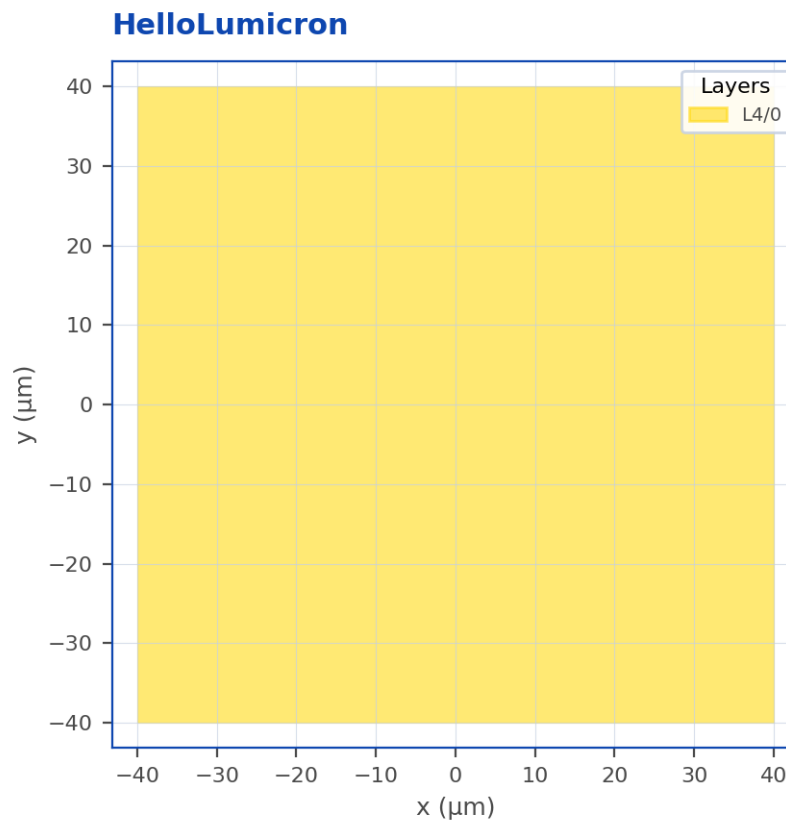


Figure 1.1.: Output of 01_hello_world.py.

i Note

The file name `HelloLumicon.gds` would have been auto-derived from the cell name if you'd called `chip.to_layout().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.

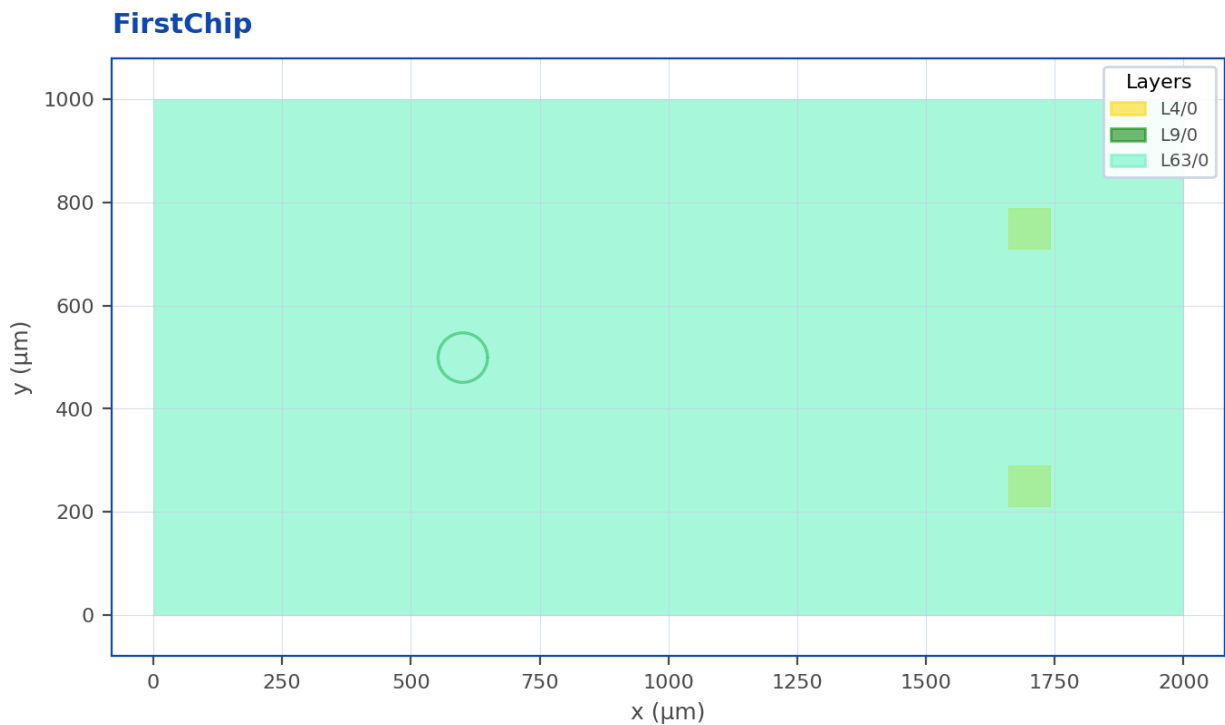


Figure 1.2.: Output of `02_first_chip.py`.

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. Calling `chip.add(pad)` twice without capturing both handles makes the second add overwrite the first in the lookup table — you'd end up moving one pad to two places, not placing two pads.

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) → (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, pin, or sub-cell.
<code>c.Place(...)</code>	Returns a placement chain for an <code>add()</code> 'd thing.
<code>c.to_layout()</code>	Finalizes the cell into a Layout.
<code>.to_gds(path?)</code>	Writes the layout to <code>.gds</code> or <code>.oas</code> .

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

Listing 1.2 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_api 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_layout().to_gds("first_chip.gds")

```

CHAPTER 2

Core concepts

Six nouns and three verbs cover almost every script you'll write.

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.
Pin	A named bbox bookmark — surfaces in the viewer's Bookmarks tab.
Layout	The compiled, exportable result of a cell tree.

Verb	What it does
add	Adds a shape, port, pin, 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`, `PIN`.
- **PascalCase** for shapes, components, and PDK objects: `Rectangle`, `Ring`, `Waveguide`, `Pad`.
- **lowercase** for chain methods: `.at()`, `.go()`, `.style()`, `.add_pin()`.

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:

Listing 2.1 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_api as lm
import lumicron_api.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_layout().to_gds("pdk_layers.gds")
```

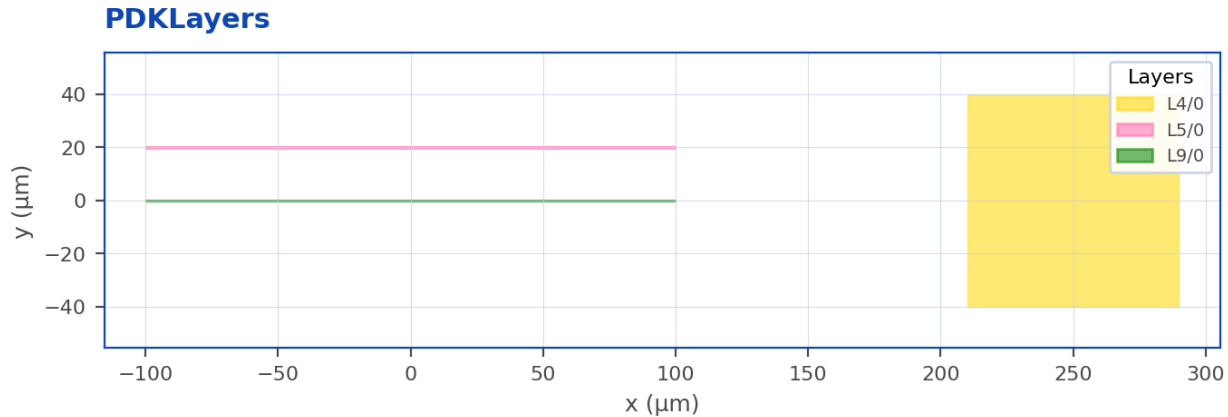


Figure 2.1.: Output of `03_pdk_layers.py`.

💡 Tip

A PDK's layers are not just numbers — they're Layer objects carrying `min_width`, `min_spacing`, and `min_radius`. The auto-router reads `min_radius` from your route's resolved layer and warns if your specified `radius=` is below it.

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)
- **Pins** (PIN — 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* — for sub-cells and arrays, the handle is what you pass to `c.Place()` and what you index for ports (`handle.port["o1"]`). For shapes you can pass either the original object or the handle to `Place()`; both work.

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.port["name"]`; on a child handle via `handle.port["name"]`.

2.5 Layouts

`c.to_layout()` finalizes a cell tree into a Layout — a frozen, exportable object. Most scripts end with:

```
return chip.to_layout()  
# or:  
chip.to_layout().to_gds("chip.gds")
```

`Layout.to_gds(path?)` writes a GDS file. Pass nothing to auto-name from the top cell; pass an explicit path to control the file name.

i Note

`to_gds()` defaults to `max_points=0` — unlimited polygon vertices. Foundries that require legacy GDS-II compatibility may need `max_points=199`; check your PDK's submission rules.

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(...)` returns a chain object. You can keep calling on it to compose a transform.

3.1 The chain

Three transforms compose, always in this order:

1. `at((x, y))` — set the placement origin.
2. `rotate_by(angle)` — rotate counter-clockwise about that origin.
3. `move_by(dx, dy)` — translate the rotated result.

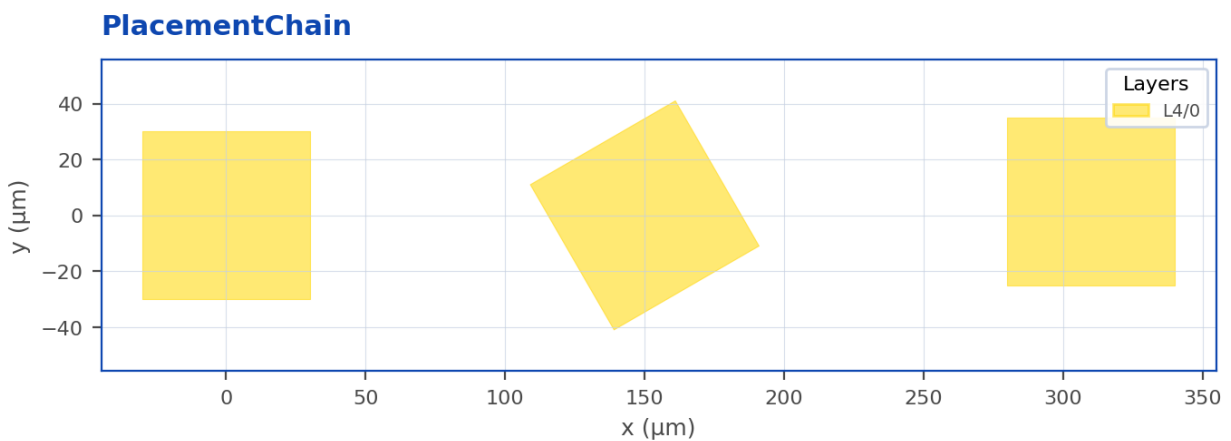


Figure 3.1.: Output of `04_placement_chain.py`.

Listing 3.1 04_placement_chain.py

```

"""The placement chain: at, move_by, rotate_by.

Every `c.Place(thing)` returns a chain. Calls compose left-to-right;
`move_by` always comes last because it's a pure translate of the result.
"""

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

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

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

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

    # 2. Place + rotate 30°. The rotation is about the placement origin.
    c.add(pad)
    c.Place(pad).at((150, 0)).rotate_by(30)

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

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

```

 Warning

`move_by` must come last. Internally it's a pure translate of the already-transformed result; placing it before `rotate_by` would rotate your translation along with the geometry, which is almost never what you want.

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.

You can also align directly to another handle:

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

3.3 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.

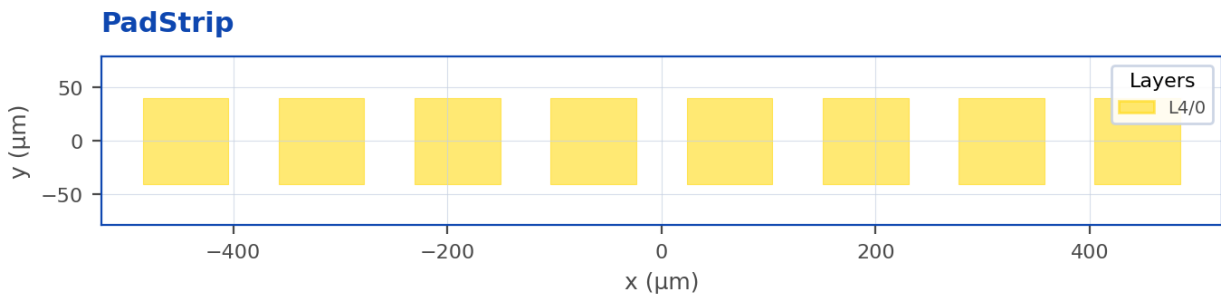


Figure 3.2.: Output of 05_arrays.py.

pitch is (dx, dy). Use (127, 0) for a horizontal strip, (0, 50) for a vertical column, or both nonzero for a parallelogram tiling. For 2D rectangular arrays, place two ARRAYS nested.

3.4 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))
```

Listing 3.2 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 lumicron_api as lm
import lumicron_api.pdks.elyon_demo.all as pdk

@lm.pcell
def Pad():
    """A single 80 × 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():
    """1×8 linear pad array on a 127 μm 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_layout().to_gds("pad_strip.gds")

```

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

3.5 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.port["mod_i1"] and chip.port["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.

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:

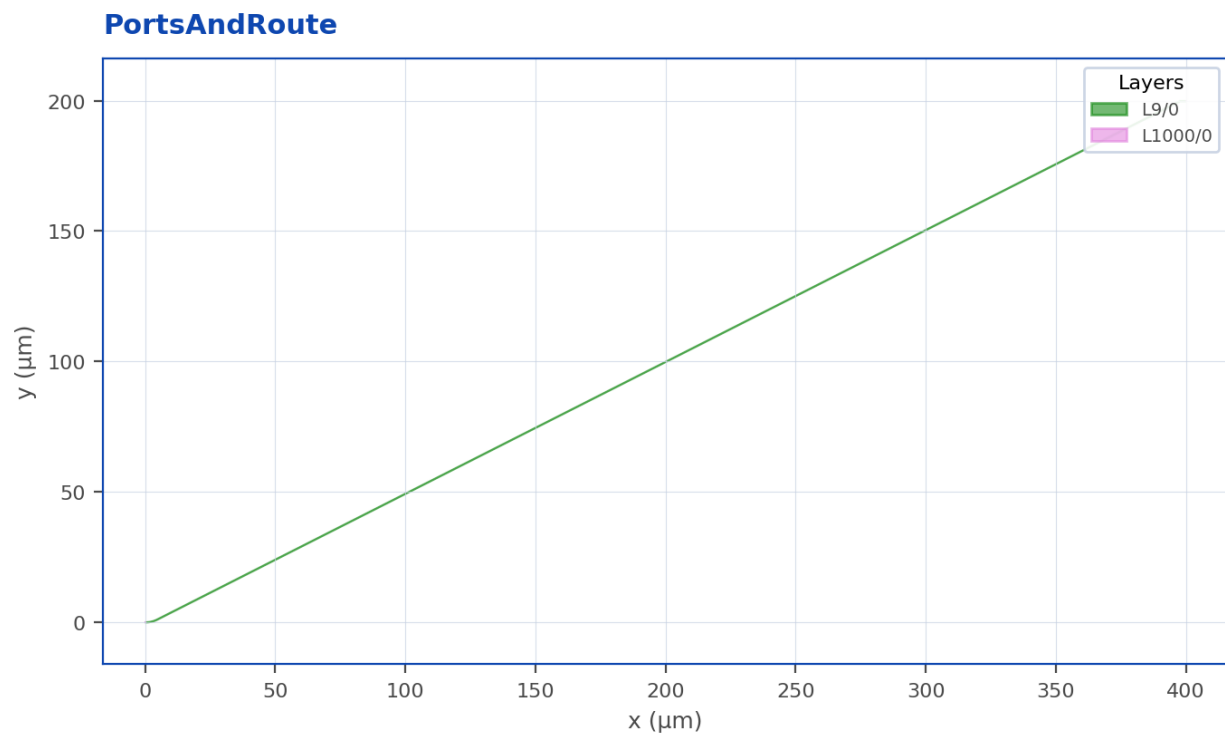


Figure 4.1.: Output of `06_ports_and_route.py`.

Listing 4.1 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_api as lm
import lumicon_api.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.port["in"], c.port["out"], radius=10)
    return c

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

```

Route(a, b, radius=10) 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=:



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

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

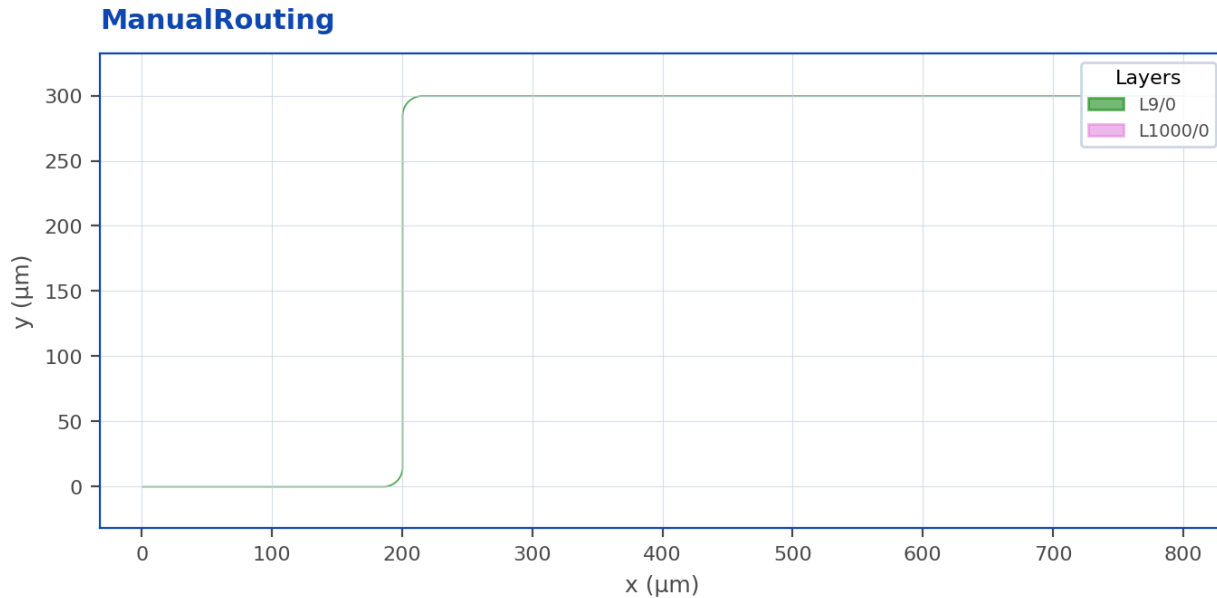


Figure 4.3.: Output of `08_manual_routing.py`.

- `.go(direction, by=...)` advances by that many μm .
- `.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 RouteBundle

To route many parallel channels with a single call:

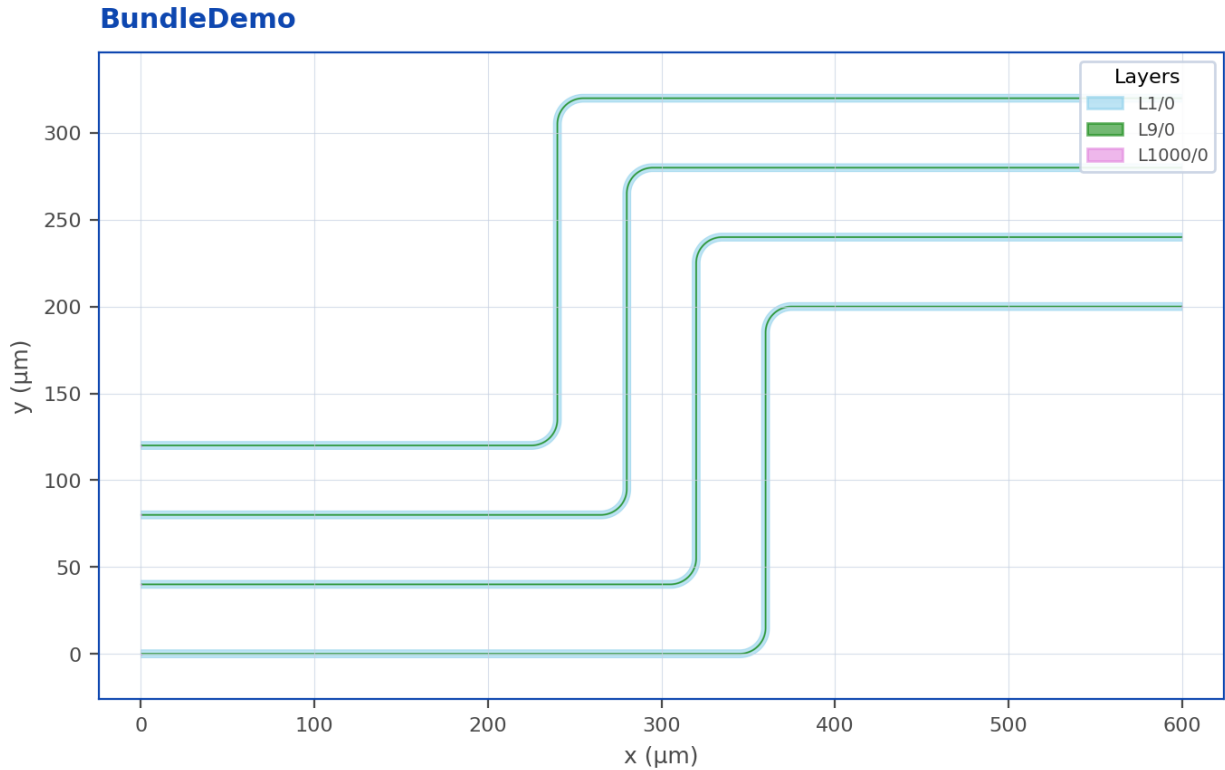


Figure 4.4.: Output of `10_route_bundle.py`.

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

- **Direct mode** (default — no `style=`, no `.go()`): 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()`): 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(...)`): drive the spine yourself, leg by leg. Use this when the auto-spine would clip an obstacle or you need a non-shortest route.

Use `.taper(...)` and `.match_lengths(...)` on the bundle chain to broadcast tapers or length-equalize all channels at once. By default `.match_lengths()` rebuilds every channel as an independent trombone (see *Bundle PLM* below) — no spine required. If you've steered the bundle with `.go()`, length matching falls back to per-channel meanders on the user-driven spine.

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:

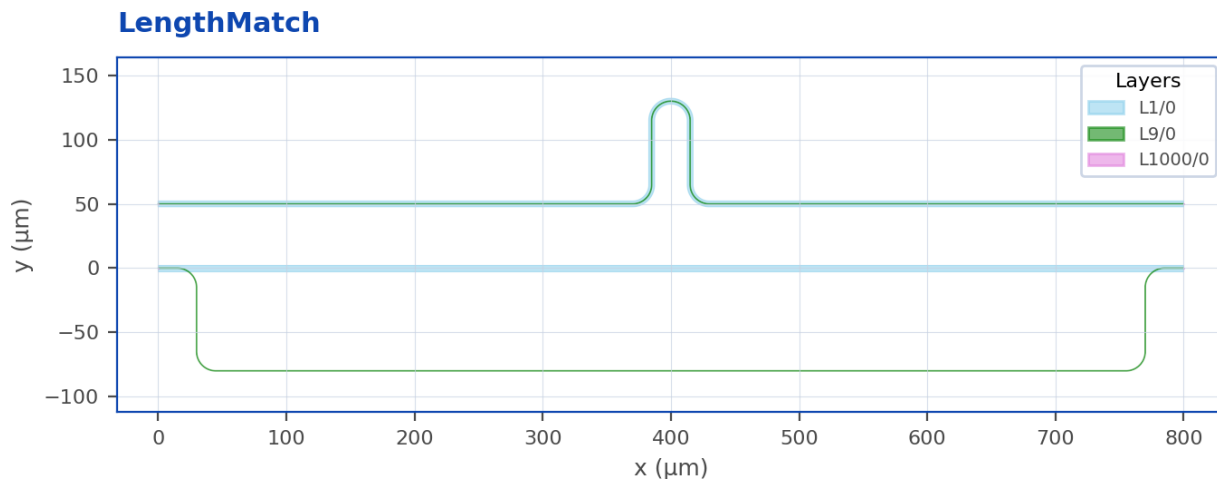


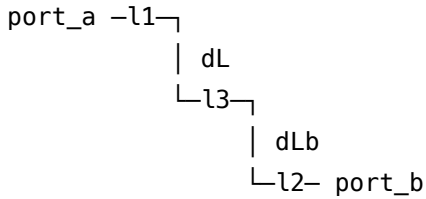
Figure 4.5.: Output of `11_length_matching.py`.

Optional knobs:

- `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.

4.5.1 Bundle PLM — rectangular detour

When you call `.match_lengths()` on a bundle that hasn't been steered with `.go()`, 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 δ 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.RouteBundle(ports_a, ports_b, spacing=, radius=) \
    .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 are matched by sorting both port lists by their transverse coordinate.

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.RouteBundle(pa, pb, spacing=, radius=).match_lengths(mirror=True)

# Two channels up, two channels down — split bundle into stacks
chip.RouteBundle(pa, pb, spacing=, radius=) \
    .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`.

`.go()`-driven bundles bypass this path — you've manually steered the spine, so length matching uses a separate per-channel meander algorithm on the longest spine leg.

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.

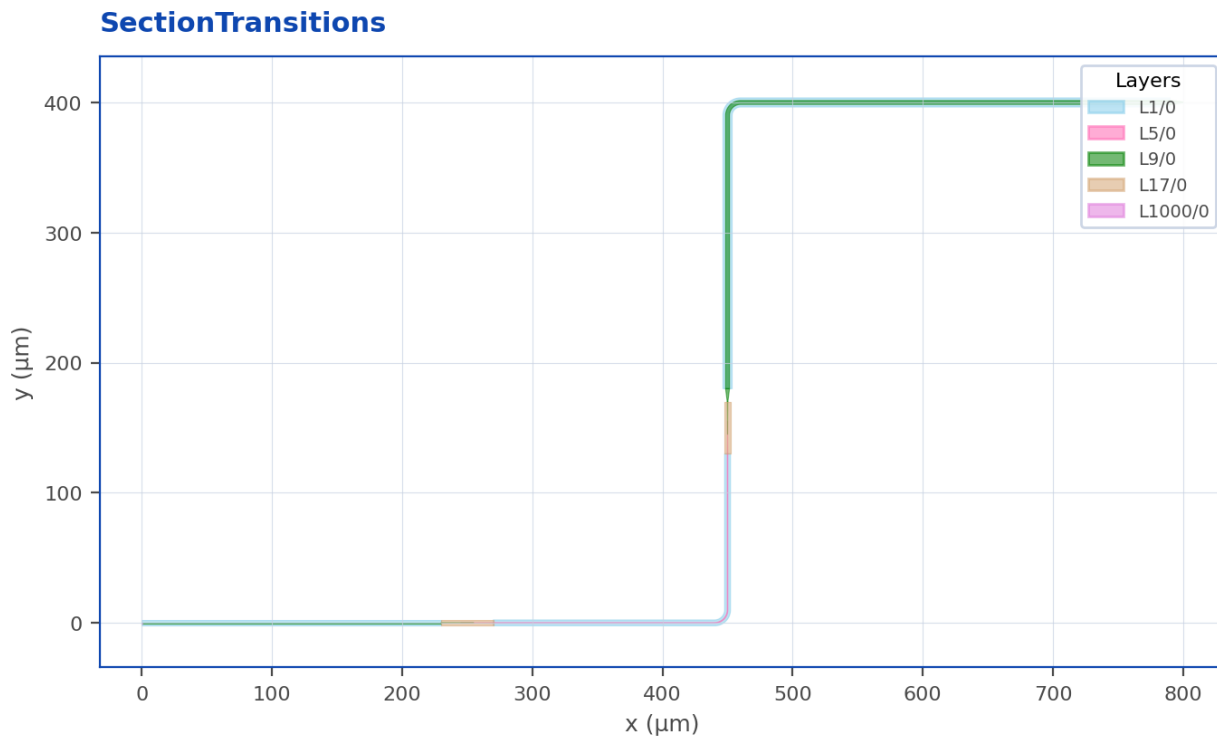


Figure 4.6.: Output of `09_section_transitions.py`.

What's happening in the example:

1. **Section 1** — silicon strip, inherited from `port_a.route_profile`.
2. `.style(rp=pkg.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 `rp=`, `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

Pass `clearance=N` to ask the router to detour around in-cell obstacles by $N\ \mu\text{m}$:

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

The avoidance pass works on Manhattan, direct, and S-bend routes, and respects the bend radius (every detour leg is at least $2 \cdot \text{radius}$ long). Source and destination cells are auto-excluded from the obstacle set.

Listing 4.2 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_api as lm
import lumicron_api.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.port[f"{style}_in"],
            c.port[f"{style}_out"],
            radius=20,
            style=style,
        )

    return c

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

```

Listing 4.3 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_api as lm
import lumicron_api.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.port["in"], c.port["out"], radius=15)
     .go("E", by=200)
     .go("N", by=300)
     .go("E", to=c.port["out"].position[0]))
    return c

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

```

Listing 4.4 10_route_bundle.py

```

"""RouteBundle – route N parallel channels with one call.

Three modes, picked by what you pass:

    - `RouteBundle(...)` (default style)          → direct: per-channel,
                                                    no coordination.
    - `RouteBundle(..., style="manhattan")`        → auto-spined manhattan
                                                    bundle; the router
                                                    picks the shortest
                                                    path from ports_a's
                                                    centroid to ports_b's.
    - `RouteBundle(...).go("E", by=...)`          → 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_api as lm
import lumicron_api.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.port[f"in{i}"] for i in range(n)]
    ports_b = [c.port[f"out{i}"] for i in range(n)]
    c.RouteBundle(ports_a, ports_b, spacing=pitch, radius=15,
                  style="manhattan")

    return c

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

```

Listing 4.5 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_api as lm
import lumicron_api.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.port["a0"], c.port["b0"], radius=15)
           .jog("S", by=80))

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

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

```

Listing 4.6 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_api as lm
import lumicron_api.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.port["in"], c.port["out"]))
    # Section 1 – silicon strip (default from port_a.route_profile).
    .go("E", by=250)
    # Section 2 – switch to silicon-nitride strip.
    .style(rp=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.port["out"].position[1])
    .go("E", to=c.port["out"].position[0]))
    return c

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

```

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:

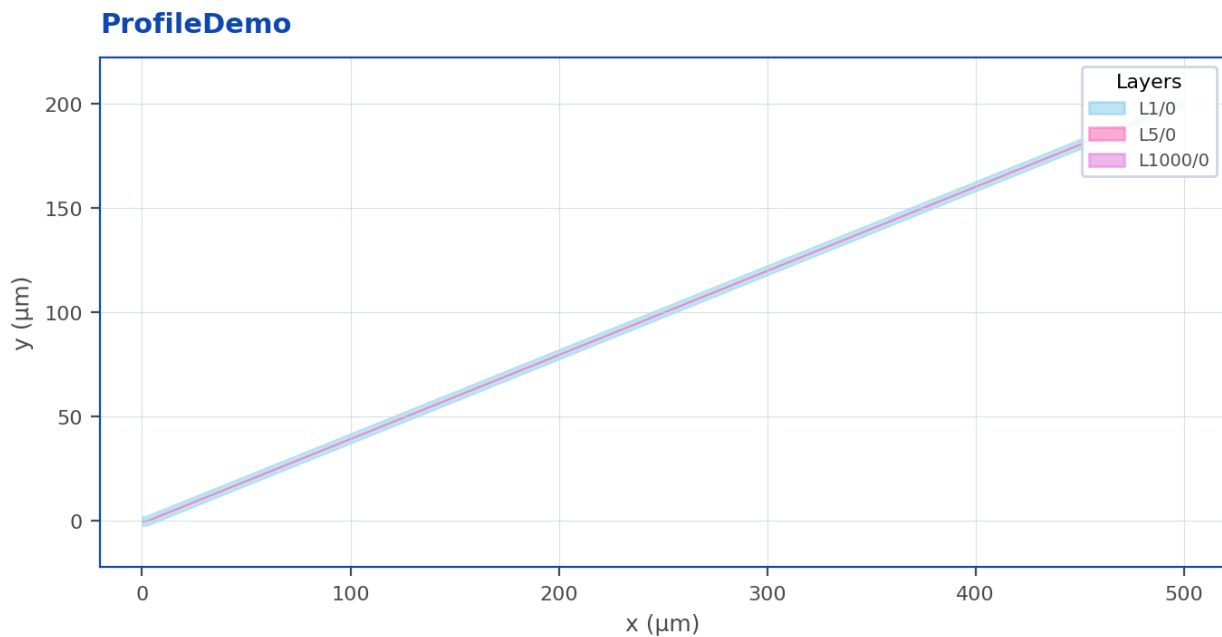


Figure 5.1.: Output of `12_route_profiles.py`.

Listing 5.1 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_api as lm
import lumicron_api.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.port["in"], c.port["out"])
    return c

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

```

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.port["o1"], modulator.port["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=...` / `width=...` override the profile’s defaults for this route only.

```
# Use the profile but force a tighter bend radius.
chip.Route(a, b, radius=8)

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

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

5.4 Auto layer transitions

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

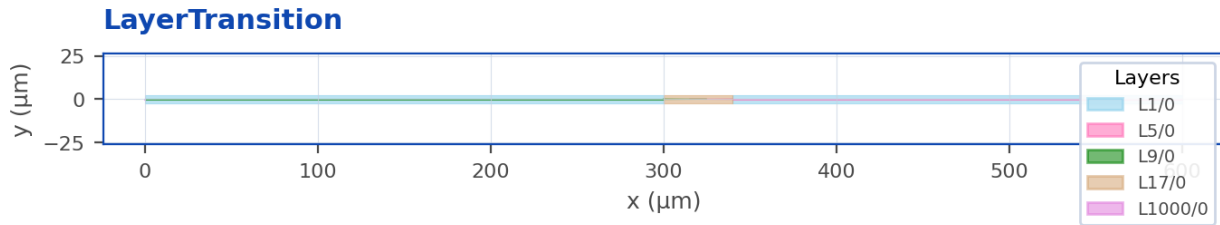


Figure 5.2.: Output of 13_layer_transition.py.

`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(rp=...)` 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.

Listing 5.2 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_api as lm

import lumicron_api.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.port["in"], c.port["out"], transition_offset=300)

```

CHAPTER 6

Pins

A **pin** 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. Pins are also what the design tool's auto-test routines look for to find devices to characterize.

Pins are not GDS labels (those are `.label()`). They're metadata that travels alongside the layout, written to a sidecar JSON when you save a `.lum` script project.

6.1 Two ways to declare a pin

1. **Attached to a placement chain** (`.add_pin(name=...)`) — the pin'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.PIN("name", at=(x, y)))`) — for arbitrary annotations like alignment marks or zoom-to-region targets.

6.2 Groups

Pass `group="..."` to bind multiple pins into a `PinGroup`. 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_pin(name=f"mod_{i}", group="modulators")
```

All pins sharing the same `group=` end up in a single `PinGroup` after `to_layout()`.

6.3 Notes

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

```
chip.add(lm.PIN(
    "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 pins on placements

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

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

Pass pin_name="..." to override. This shorthand is equivalent to:

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

6.5 Empty names are rejected

lm.PIN("", at=(0, 0)) raises ValueError. Same for whitespace-only names and add_pin(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.

Listing 6.1 14_pins.py

```

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

A pin 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.PIN("name", at=(x, y)))`
    2. Attached to a placement chain: `.add_pin(name="...")`
"""

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

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

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

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

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

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

if __name__ == "__main__":
    PinDemo().to_layout().to_gds("pin_demo.gds")

```

Part III.

Working with PDKs

CHAPTER 7

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 verified geometry and properly-attached ports (EdgeCouplerSi, NxM_MMI, PhotonicElevator, modulators, ...).
- A transition registry so cross-layer routes auto-insert the right PCell.

lumicron_api ships with elyon_demo, an internal PDK we use throughout this guide. Foundry-specific PDKs are distributed as .lumpdk packages installable through the Lumicron app or via pip install.

7.1 Importing

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

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

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

In a Lumicron script project, the active PDK is auto-imported and bound to the name pdk — your scripts can skip the import line.

7.2 Discovering what's in a PDK

All three namespaces 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`).

7.3 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

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

The `@pcell` decorator hashes the non-default arguments into the cell name, so two coupler variants don't collide in GDS. Repeated calls with identical params return the same cached cell — `c.add()` then creates two independent instance refs.

7.4 Component ports

Every PDK component publishes ports with a `route_profile` attached. That means `c.Route(coupler.port["o1"], modulator.port["i1"])` — no profile, no radius — just works. The route stamps the right layers, pulls the right bend radius, and inserts a layer transition if the two ports cross layers.

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.

7.5 Installing additional PDKs

Foundry-specific PDKs ship as `.lumpdk` files (zip archives with a `manifest.json` and Python source). Two install paths:

- **In the Lumicron Mac app:** Settings ▢ PDKs ▢ Install. The app extracts the source into `~/Library/Application Support/Lumicron/UserPDKs/` and registers it on the `lumicron_api.pdks` import path.
- **By hand from Python:** `pip install <wheel>` if the PDK ships as a wheel.

After install, import `lumicron_api.pdks.<name>.all` as `pdsk` works exactly like the demo PDK.

7.6 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(...)`).

Part IV.

Cookbook

CHAPTER 8

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`.

8.1 Mach-Zehnder interferometer

A balanced 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.

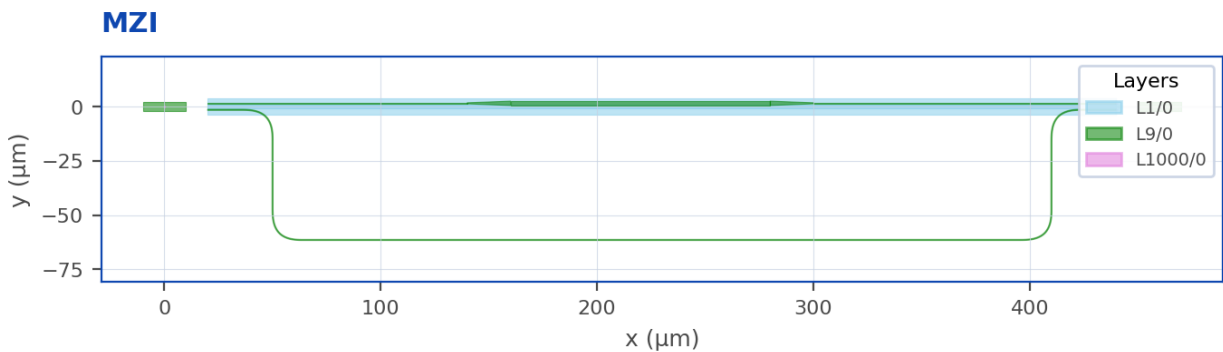


Figure 8.1.: Output of `15_mzi.py`.

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 `i1` faces left, matching the arm-end geometry.
- `.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.

To turn this into an unbalanced MZI, replace the bottom arm's `.jog()` with a `.match_length_to(top, delta= $\Delta$ )` for an explicit ΔL .

8.2 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.

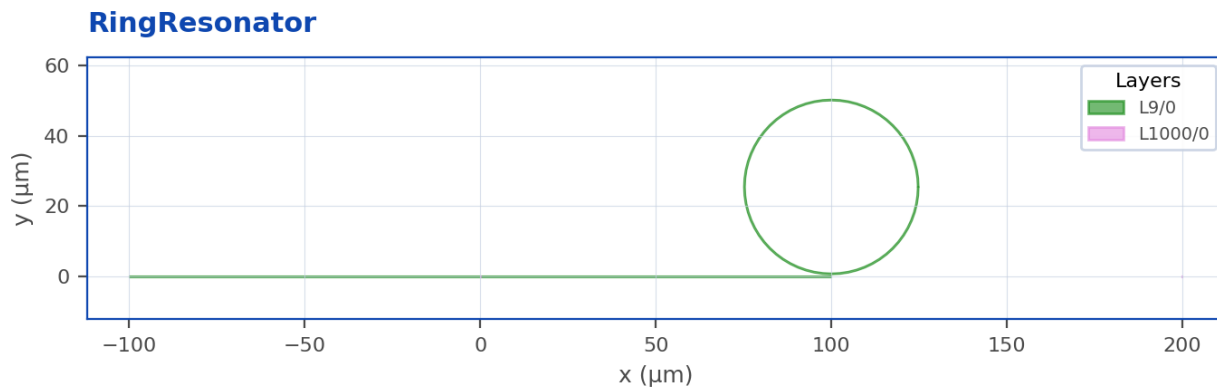


Figure 8.2.: Output of `16_ring_resonator.py`.

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.

8.3 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.

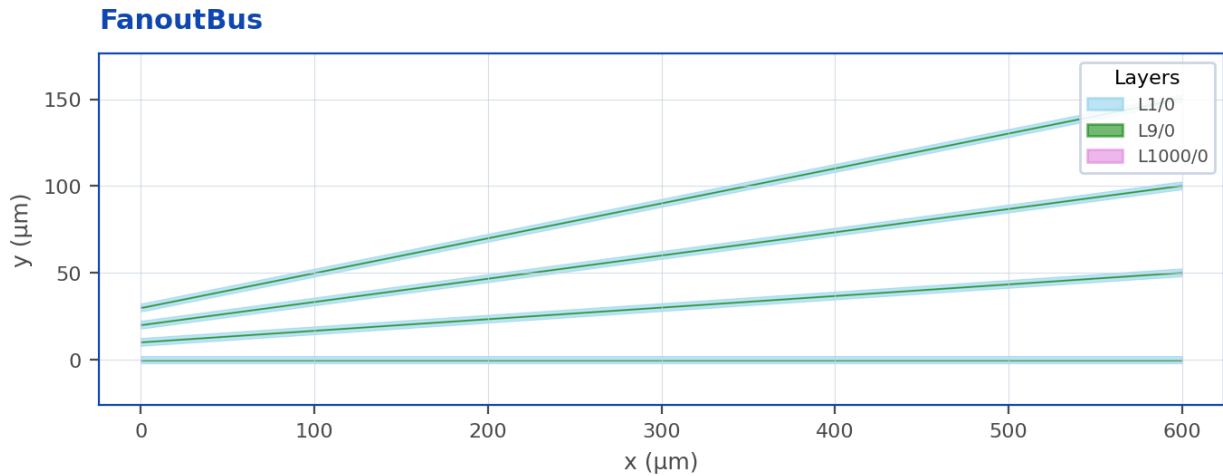


Figure 8.3.: Output of `17_fanout_bus.py`.

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. `RouteBundle` 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.

Listing 8.1 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() and is then length-matched back to the top.
"""

import lumicron_api as lm
import lumicron_api.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.port["o1"], k.port["o2"], radius=15, bend="euler")
     .taper(offset=120, width=2.0, length=20)
     .taper(offset=120, width=0.5, length=20))

    # Bottom arm - jog south by `delay` to make it longer, then match.
    c.Route(s.port["o2"], k.port["o1"], radius=15, bend="euler") \
        .jog("S", by=delay)
    return c

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

```

Listing 8.2 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_api as lm
import lumicron_api.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_layout().to_gds("ring_resonator.gds")

```

Listing 8.3 17_fanout_bus.py

```

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

A 4-channel bundle entering at 10  $\mu\text{m}$  pitch and leaving at 50  $\mu\text{m}$ . The
auto-fanout absorbs the pitch difference into a U-shape on each side;
the spine in between stays parallel and tight.
"""
import lumicron_api as lm
import lumicron_api.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.port[f"in{i}"] for i in range(n)]
    b = [c.port[f"out{i}"] for i in range(n)]
    c.RouteBundle(a, b, spacing=pitch_in, radius=15)
    return c

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

```

Cheat sheet

A one-page reference. Skim this once before reading anything else.

Boilerplate

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

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

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

Placement

Chain method	Effect
<code>.at((x, y))</code>	Set placement origin
<code>.using("W"/"E"/"N"/"S"/"NE"/...)</code>	Use a child anchor instead of its origin
<code>.rotate_by(deg)</code>	Rotate CCW
<code>.move_by(dx, dy)</code>	Translate (always last)
<code>.right_of(other), .left_of(other)</code>	Anchor-to-anchor align
<code>.above(other), .below(other)</code>	Anchor-to-anchor align

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.port["name"] / handle.port["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.

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="auto",        # "auto" | "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=None, # auto layer-transition position
            transitions=None)    # custom TransitionRegistry
```

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=, rp=, 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

RouteBundle

```
chip.RouteBundle(ports_a, ports_b,
                 spacing=,           # bundle pitch (μm)
                 radius=,
                 fanout="manhattan", # "manhattan" | "sbend"
                 length_match=False) # True | "longest" | float | list
```

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

Pins

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

Output

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

Useful introspection

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

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`.

Errors

`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 <code>PCell</code> 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.

`GeometryError`

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

ValueError from `lm.PIN(...)` / `add_pin(...)`

- "requires a non-empty name" — empty / whitespace-only name.
- "requires either `at=` or `bbox=`" — neither given.
- "bbox corners are inverted" — `xmax < xmin` or `ymax < ymin`.

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).

Warnings

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

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.
"radius ... below layer <code>min_radius</code> "	<code>radius=</code> is below the route layer's <code>min_radius</code> rule.	Increase <code>radius=</code> , or change to a layer with a tighter rule.
"bridge taper auto-inserted at <code>port_b</code> "	Final width didn't equal <code>port_b.width</code> .	Add an explicit <code>.end_taper(...)</code> or finish with <code>.width(port_b.width)</code> .

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 <code>amplitude=</code> produced a length that misses the target by $> 1\ \mu\text{m}$.

ProfileWarning

Message contains	Cause
"radius=... overrides the profile's radius"	You passed <code>radius=</code> <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.

Suppressing warnings

Standard Python `warnings.filterwarnings` works:

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

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

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.RouteBundle(...)`.

Cell — A reusable hierarchical container of geometry, ports, sub-cells, and pins. `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.

Pin — 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.

PinGroup — A collection of pins 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 ↔ SiLN). Registered in a `TransitionRegistry`; the router consults it when a route crosses layers.