



PHOTONIC LAYOUT, FROM PYTHON.

PDK Authoring Guide

Building layers, profiles, and components for the `lumicron_api`

Version	0.5.0
Released	May 2026
Publisher	Elyon Semiconductor

Lumicron PDK Authoring Guide

© 2026 Elyon Semiconductor. All rights reserved.

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

This guide describes how to package process-specific layers, components, and route profiles for the `lumicron_api` ecosystem. For end-user scripting, see the companion *Lumicron API User Guide*.

Code listings are extracted from the `pdk_authoring_docs/examples/` directory of the source tree and are tested at build time.

Photonic Layout, From Python.

Table of contents

Welcome	1
What “PDK” means in this guide	1
Preface	3
Who this is for	3
What’s covered	3
A quick orientation	3
Conventions	4
I. Anatomy	5
1. Anatomy of a PDK	6
1.1. How an end user sees it	7
1.2. What ships in the package vs the manifest	7
1.3. What this guide does not cover	7
II. Building blocks	9
2. Layers and design rules	10
2.1. The minimum	10
2.2. Design rules	10
2.3. Naming	11
2.4. Datatypes	11
2.5. Programmatic introspection	12
3. The physical stack	14
3.1. A complete stack	14
3.2. Sellmeier coefficients	15
3.3. Layer order	16
3.4. When the stack matters at runtime	16
4. Route profiles	18
4.1. The decorator	18
4.2. A minimal profile — single core	19
4.3. Continuous strokes — <code>p.layer()</code>	20

4.4. Periodic features — <code>p.tooth()</code>	21
4.5. Profile collections	22
4.6. Which knob the route reads	22
4.7. Profile design patterns	23
5. Components	27
5.1. A simple component	27
5.2. Publishing ports with route profiles attached	28
5.3. What “good ports” look like	29
5.4. Synthesizing geometry	29
5.5. Promoting child ports	29
5.6. Component design patterns	30
6. Layer transitions	33
6.1. Registering a transition	33
6.2. The factory contract	34
6.3. What a transition cell needs	34
6.4. Where transitions appear in a route	34
6.5. When the registry doesn’t have an entry	35
6.6. What ships with the API	35
III. Packaging & distribution	37
7. Packaging — the <code>.lumpdk</code> format	38
7.1. Anatomy	38
7.2. The manifest	38
7.3. Building a <code>.lumpdk</code>	39
7.4. Where it lands after install	40
7.5. Versioning the bundle	40
7.6. What can go wrong	41
8. Publishing	42
8.1. Versioning	42
8.2. Publishing to PyPI	43
8.3. Publishing a <code>.lumpdk</code>	43
8.4. What to put in your README	44
8.5. CI / release pipeline	44
8.6. Marketing your PDK	44
IV. Cookbook	46
9. Cookbook — a minimal PDK from scratch	47
9.1. Splitting it into a package	48
9.2. What this minimal PDK is missing	48
9.3. Verifying it works	49

Appendices	51
Manifest schema	51
Required fields	51
Recommended fields	51
The layers array	52
Optional fields the app understands	52
Optional fields ignored by the current app	53
Example — complete manifest	53
Validation	53
Glossary	55

Welcome

This is the **PDK Authoring Guide** for `lumicron_api`. It tells you how to package a process design kit — layers, route profiles, components, layer transitions — that end users can import and use without thinking.

If you're an end user looking to *write* layouts (not author a PDK), the companion **Lumicron API User Guide** is what you want.

What “PDK” means in this guide

A PDK in `lumicron_api` is, at minimum, four things:

1. A **layer table** — every named GDS layer your process exposes, with design rules attached.
2. A **layer stack** — the physical reality (thicknesses, dispersive indices) for simulators.
3. A **route profile** for every cross-section a router should know how to draw.
4. A **component library** — `@pcell`-decorated factories whose ports already carry the right route profiles.

Optional but encouraged:

- A **transition registry** so cross-layer routes auto-insert the right PCell.
- A **.lumpdk package** with a manifest, so the Lumicron Mac app can install it as a process.

When all five are in place, an end user's script is just `import lumicron_api.pdks.<your_pdk>.all` as `pdk` followed by component placement and routing — every kwarg they don't pass picks up the right default from the PDK.

Tip

The single biggest investment a PDK author makes is **route profiles attached to component ports**. When every component publishes ports that already carry their cross-section, end-user scripts shrink dramatically: `chip.Route(coupler.port["o1"], modulator.port["i1"])` does the right thing with no kwargs.

Every code listing in this book is a real, executable file in `pdk_authoring_docs/examples/`. The build script runs them, regenerates figures, and rebuilds this PDF.

Preface

Who this is for

You're publishing a process for other people to write layouts against. Maybe it's your foundry's PDK, your in-house process for a research group, an open-source PDK port, or a vertical-specific layer set you want to share publicly. Either way, you're the layer between the process spec and the end user, and you want their experience to feel polished.

You should already be comfortable with `lumicron_api` as an end user. If not, read the *Lumicron API User Guide* first — this guide assumes you know what `c.Route(...)`, `@pcell`, and the placement chain do.

What's covered

- **Anatomy** (Chapter 1): the four parts of a PDK and how they relate.
- **Building blocks** (Chapters 2–6): layers and rules, the physical stack, route profiles, components, and transitions — one chapter each.
- **Packaging** (Chapter 7): the `.lumpdk` format and how the app installs it.
- **Publishing** (Chapter 8): versioning, distribution, the marketplace.
- **Cookbook** (Chapter 9): a complete minimal PDK in one file you can adapt.
- **Appendices**: manifest schema, glossary.

A quick orientation

Compared to other layout DSLs:

- The **route profile** is `lumicron_api`'s answer to `gdsfactory`'s `cross_section`. The mental model is similar but the implementation differs in two ways relevant to PDK authors. (1) Periodic features (gratings, distributed taps, slot-mode rails) are first-class via `p.tooth(...)`, not bolted on after. (2) Profiles attach to ports, so a route inherits the cross-section from its endpoints rather than being recomputed at every call site.

- **@pcell PCells** behave like IPKISS PCells — same parameters □ same cell name □ same GDS cell definition. The differences from IPKISS / KFactory are mostly cosmetic (decorator vs class, kwargs vs param objects).
- **Layer transitions** are explicit and named. Each (layer_a, layer_b) pair has at most one factory in the registry. The router doesn't try to guess; if a pair isn't registered, it errors.

Conventions

The same callout styles as the User Guide:

Note

A clarification about why something is the way it is.

Tip

A pattern that pays back consistently as your PDK grows.

Warning

A trap to know about before you hit it.

Part I.

Anatomy

CHAPTER 1

Anatomy of a PDK

A `lumicron_api` PDK is a Python package — by convention, `lumicron_api.pdks.<name>` — exposing four namespaces:

Namespace	Type	Purpose
LAYER	LayerTable	Named layers + design rules.
STACK	LayerStack	Physical thicknesses + dispersive indices for simulators.
RP (top-level)	RouteProfiles @pcell factories	Cross-section recipes routes can adopt. Component library — <code>EdgeCouplerSi</code> , <code>Pad</code> , <code>NxM_MMI</code> , ...
TRANSITIONS	TransitionRegistry	Optional. Maps <code>(layer_a, layer_b)</code> → bridge PCell.

The conventional file layout:

```
lumicron_api/pdks/<name>/
  __init__.py      # re-exports the four namespaces
  all.py           # convenience: `import ... .all as pdk`
  layers.py        # LAYER, STACK
  profiles.py      # RP, plus @route_profile defs
  components.py    # @pcell factories
  transitions.py    # TRANSITIONS, registers into default_transitions()
```

Each file is small. The discipline is keeping them small — when a PDK gets big, prefer adding new modules (`pcells/`, `tests/`) over fattening the four canonical files.

1.1 How an end user sees it

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

pdk.LAYER.SILC          # named layer with design rules
pdk.RP.silc_strip       # route profile
pdk.EdgeCouplerSi(...)  # component factory
```

The `.all` re-export gives end users the convention of one short import. Implementing it is a single file:

```
# lumicron_api/pdks/<name>/all.py
from lumicron_api.pdks.<name>.layers import LAYER, STACK
from lumicron_api.pdks.<name>.profiles import RP
from lumicron_api.pdks.<name>.transitions import TRANSITIONS
from lumicron_api.pdks.<name>.components import * # noqa
```

The transitions import — even though TRANSITIONS itself is rarely referenced by name — is what registers the transitions into the process-wide default registry. Without it, end users would have to remember to import the module or pass `transitions=` on every cross-layer route. With it, things just work.

1.2 What ships in the package vs the manifest

Two questions that come up:

- **“Should the layer table go in Python or in JSON?”** Both. The Python `LayerTable` is the source of truth — the router reads it, components reference it. The `.lumpdk` manifest exposes a JSON layer map so the Mac app can render colors / names without importing the Python source. They’re generated from each other; in `lumicron_api`, `LayerTable.to_manifest()` produces the JSON.
- **“Should the layer stack ship?”** Yes when you have it. End-user scripts that drive simulators read it. PDKs without a known stack can ship Python-only and add the stack later — components and routing don’t depend on it.

1.3 What this guide does not cover

- **DRC rules beyond the per-layer minimums.** The full Lumicron DRC system is documented separately in `Docs/DRC_DESIGN.md` (forthcoming).
- **Verification / LVS.** Same — separate doc.

- **Foundry-confidential details.** Whatever you ship in your PDK is up to you and your foundry agreement; this guide is process-agnostic.

Part II.

Building blocks

CHAPTER 2

Layers and design rules

Layers are the foundation. Every component, every route, every shape in your PDK will reference them by name, so naming and rule-tagging them well pays back on every script that uses your PDK.

2.1 The minimum

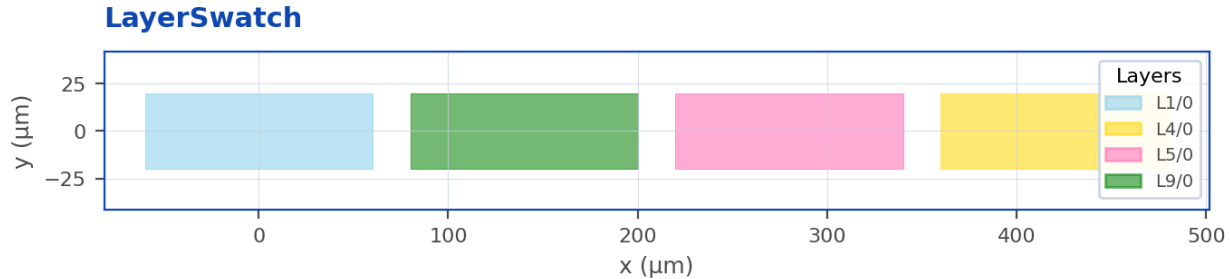


Figure 2.1.: Output of `01_define_layers.py` — one swatch per layer.

Two things this example introduces:

1. `lm.Layer(layer, datatype, ...)` — a `LayerSpec` with attached design rules. Use it anywhere the API takes a `LayerSpec`.
2. `lm.LayerTable(NAME=..., ...)` — collects layers into an object with attribute access. End users write `pdk.LAYER.SILC`; you write `getattr(LAYER, name)` only when you must iterate.

2.2 Design rules

Layer carries three rule fields the API actively consults:

Field	Where it's used
<code>min_width</code>	Validation (DRC), warned when shapes are stamped narrower.
<code>min_spacing</code>	Same, for adjacent shapes on the same layer.
<code>min_radius</code>	The router compares your <code>radius=</code> against this and warns if too tight.

Add them as you have them — partial coverage is fine. Layers without rules just don't trigger checks.

```
SILC = lm.Layer(9, 0,
                color="#008000",
                min_width=0.2, # 200 nm minimum body width
                min_spacing=0.2, # 200 nm minimum spacing
                min_radius=5.0) # 5 μm minimum bend radius
```

Tip

The `color` field is a single source of truth — it shows up in the Lumicron Mac viewer's layers panel, in the figure renderer for documentation, and (when exported via `LAYER.to_manifest()`) in the `.lumpdk` JSON. Use the same color across all three so users don't get confused by re-renderings.

2.3 Naming

Stick to short, all-caps names that describe the *layer*, not the *use*. `SILC`, `SILN`, `MTL1`, `VIA1`, `0X0P` — these compose well into longer identifiers (`SILC_strip`, `MTL1_via1_stack`). Avoid usage-coupled names like `SHALLOW_RIB` that lock the layer to one cross-section; profiles handle that.

If your foundry's docs use names like `LP1` or `F0X`, mirror those — your end users are reading the foundry datasheet alongside your PDK and dual-naming will burn them.

2.4 Datatypes

Datatypes are a foundry convention that lets you subclassify shapes on the same layer without minting new layer numbers. Common patterns:

- **drawn** = datatype 0
- **exclusion** = datatype 1
- **fill** = datatype 2
- **identification text** = datatype 99

If you publish multiple datatypes, give each a distinct entry in LAYER:

```
LAYER = lm.LayerTable(
    SILC      = lm.Layer(9, 0, ...),
    SILC_EXCL = lm.Layer(9, 1, ...),
)
```

2.5 Programmatic introspection

LAYER.print_rules() prints the table — useful in scripts users write while exploring your PDK:

```
SILC  L9/0    min_width=0.2  min_spacing=0.2  min_radius=5.0
SILN  L5/0    min_width=0.4  min_spacing=0.3  min_radius=10.0
MTL1  L4/0    min_width=1.0  min_spacing=1.0
OXID  L1/0
```

LAYER.to_layer_map() and LAYER.to_manifest() emit machine-readable forms — used by the .lumpdk packaging step (Chapter 7).

Listing 2.1 01_define_layers.py

```

"""Defining layers with design rules and a LayerTable.

Each `lm.Layer` carries a (layer, datatype) pair plus optional design
rules. A `LayerTable` collects them with attribute access – the lookup
your end-users will reach for as `pdk.LAYER.SILC`.
"""

import lumicron_api as lm

LAYER = lm.LayerTable(
    OXID = lm.Layer(1, 0, color="#87ceeb"),
    SILC = lm.Layer(9, 0, color="#008000",
                    min_width=0.2, min_spacing=0.2, min_radius=5.0),
    SILN = lm.Layer(5, 0, color="#ff69b4",
                    min_width=0.4, min_spacing=0.3, min_radius=10.0),
    MTL1 = lm.Layer(4, 0, color="#ffd700",
                    min_width=1.0, min_spacing=1.0),
)

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

    # One swatch per layer to render the palette.
    for i, name in enumerate(["OXID", "SILC", "SILN", "MTL1"]):
        rect = lm.Rectangle(x_dim=120, y_dim=40, layer=getattr(LAYER, name))
        c.add(rect)

        c.Place(rect).at((i * 140, 0))
    return c

if __name__ == "__main__":
    LAYER.print_rules()
    LayerSwatch().to_layout().to_gds("layer_swatch.gds")

```

CHAPTER 3

The physical stack

The LayerStack describes the physical reality of your process: what sits on what, how thick each layer is, and how each material refracts light. End-user scripts use it to drive simulators — mode solvers, FDTD wrappers, RCWA — that need a cross-section to work from.

If your PDK doesn't have public stack data (e.g. NDA'd foundry process), you can ship the layer table only and add the stack later. Components and routing don't depend on it.

3.1 A complete stack

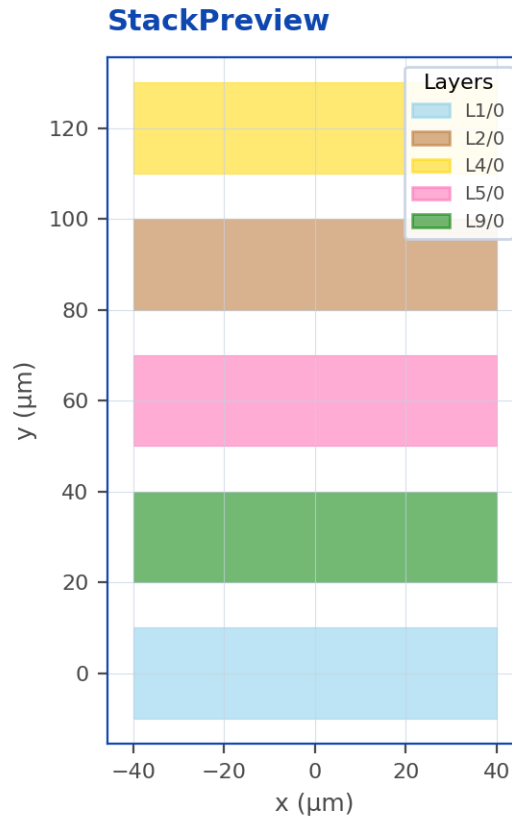


Figure 3.1.: Output of `02_layer_stack.py` — process layers as horizontal swatches.

Three building blocks:

Type	Purpose
Sellmeier	Coefficients for $n^2(\lambda) = 1 + \sum B_i \cdot \lambda^2 / (\lambda^2 - C_i)$.
StackLayer	One physical layer: name, GDS layer ref, thickness, optical model.
LayerStack	Ordered collection of StackLayers, bottom to top.

3.2 Sellmeier coefficients

The Sellmeier equation is the standard way to express dispersive $n(\lambda)$ for transparent dielectrics. `lumicron_api.Sellmeier(B=[...], C=[...])` takes the textbook coefficients in B_i / λ^2 -units. SiO₂, Si, and SiN have well-known constants — copy them from a reference (Palik, Malitson, IUPAC) so your end users get accurate dispersion.

For materials with no Sellmeier expansion (metals, doped layers, novel materials), pass $n =$ directly:

```
lm.StackLayer("metal1", layer=LAYER.MTL1, thickness=1.0, n=0.0)
```

The `n=0.0` here means “treat as opaque metal” — most simulator wrappers interpret it as a perfect electrical conductor for the purposes of mode confinement.

3.3 Layer order

Stack order matters: the first `StackLayer` sits at the bottom of the wafer, each successive layer above it. For a typical SOI photonic process:

1. BOX (buried oxide)
2. Silicon waveguide core
3. (optional) silicon nitride strip
4. Top oxide cladding
5. Metal-1
6. Inter-metal oxide
7. Metal-2

Match your foundry’s PDK datasheet exactly. Off-by-one errors here are subtle — they don’t break compilation, just mode-solver results.

3.4 When the stack matters at runtime

The router doesn’t read the stack. The shape compiler doesn’t read it either. The stack only enters the picture when:

- A user runs a simulator that consumes `pdk.STACK`.
- A user calls `pdk.STACK.cross_section(at=...)` to get the layer ordering at a specific x-coordinate (used by mode solvers).
- The Lumicron viewer renders a 3-D preview (forthcoming).

So getting the stack right is mostly a matter of accuracy, not blast radius. You can ship a working PDK with a placeholder stack and refine it as you collect material data.

i Note

The stack does **not** describe processing steps (e.g. “after litho on SILC, etch is partial”). It’s a finished-process model. Process-history modeling, when it matters (TCAD-style flows), lives outside the PDK in tooling that consumes the stack as a starting point.

Listing 3.1 02_layer_stack.py

```

"""A physical layer stack with Sellmeier dispersion.

The `LayerStack` describes the physical reality of the process – what
sits on what, how thick each layer is, and the material's dispersive
refractive index. Simulators (mode solvers, FDTD wrappers) read the
stack to set up their cross-section.
"""

import lumicron_api as lm

LAYER = lm.LayerTable(
    BOX = lm.Layer(1, 0, color="#87ceeb"),
    SILC = lm.Layer(9, 0, color="#008000", min_width=0.2,
                    min_spacing=0.2, min_radius=5.0),
    SILN = lm.Layer(5, 0, color="#ff69b4", min_width=0.4,
                    min_spacing=0.3, min_radius=10.0),
    CLAD = lm.Layer(2, 0, color="#aac7e8"),
    MTL1 = lm.Layer(4, 0, color="#ffd700"),
)

# Sellmeier coefficients describe n(λ) for a material. The form is
#  $n^2(\lambda) - 1 = \sum B_i \cdot \lambda^2 / (\lambda^2 - C_i)$ .
SiO2 = lm.Sellmeier(
    B=[0.6962, 0.4079, 0.8975],
    C=[0.0684**2, 0.1162**2, 9.8962**2],
)
Si = lm.Sellmeier(
    B=[10.6684, 0.0030, 1.5413],
    C=[0.3015**2, 1.1348**2, 1104.0**2],
)
SiN = lm.Sellmeier(B=[2.8939], C=[0.1397**2])

STACK = lm.LayerStack(
    lm.StackLayer("BOX", layer=LAYER.BOX, thickness=2.0, sellmeier=SiO2),
    lm.StackLayer("core_si", layer=LAYER.SILC, thickness=0.22, sellmeier=Si),
    lm.StackLayer("core_sin", layer=LAYER.SILN, thickness=0.4, sellmeier=SiN),
    lm.StackLayer("clad", layer=LAYER.CLAD, thickness=2.0, sellmeier=SiO2),
    lm.StackLayer("metal1", layer=LAYER.MTL1, thickness=1.0, n=0.0),
)

@lm.pcell
def StackPreview():
    """A layout that exposes the stack's process layers as a strip."""
    c = lm.CELL("StackPreview")
    for i, name in enumerate(["BOX", "SILC", "SILN", "CLAD", "MTL1"]):
        rect = lm.Rectangle(x_dim=80, y_dim=20, layer=getattr(LAYER, name))

        h = c.add(rect)
        c.Place(h).at((0, i * 30))
    return c

if __name__ == "__main__":
    LUMICRON · PDK Authoring Guide
    StackPreview().to_layout().to_gds("stack_preview.gds")

```

CHAPTER 4

Route profiles

A route profile is the cross-section recipe a router consults when it stamps a waveguide. It describes:

- One or more **continuous strokes** — the core, the claddings, exclusion zones, slot rails — each at a given width and offset.
- Optionally, **periodic features** — grating teeth, periodic claddings, electrical stitching — laid down at a fixed period along the route.

Once a profile exists, end users stop thinking about per-stroke geometry. They attach the profile to a port; every route off that port inherits it. **This chapter is the most important one in this guide** because attaching profiles to component ports is the single biggest UX investment a PDK author makes.

4.1 The decorator

```
import lumicron_api as lm

@lm.route_profile(port_width=0.5, radius=10.0, radius_min=5.0)
def my_profile(p):
    p.layer(LAYER.SILC, width=p.port_width)
```

Three decorator kwargs:

Kwarg	Purpose
port_width	Default PORT.width when callers don't override. Available as p.port_width inside the body.
radius	Default bend radius. Routes inherit it unless they pass an explicit radius=.

Kwarg	Purpose
radius_min	Minimum allowed bend radius — the router warns below this even if $\text{radius_min} < \text{radius}$. Defaults to radius.
port_layer	Optional fallback for <code>PORT.layer</code> . Defaults to the first layer declared in the body.
name	Optional explicit name. Defaults to the function name.

The decorator returns a `RouteProfile` object. Pass it as `route_profile=` on a `PORT(...)` call, or as `profile=` on a `Route(...)` call.

4.2 A minimal profile — single core

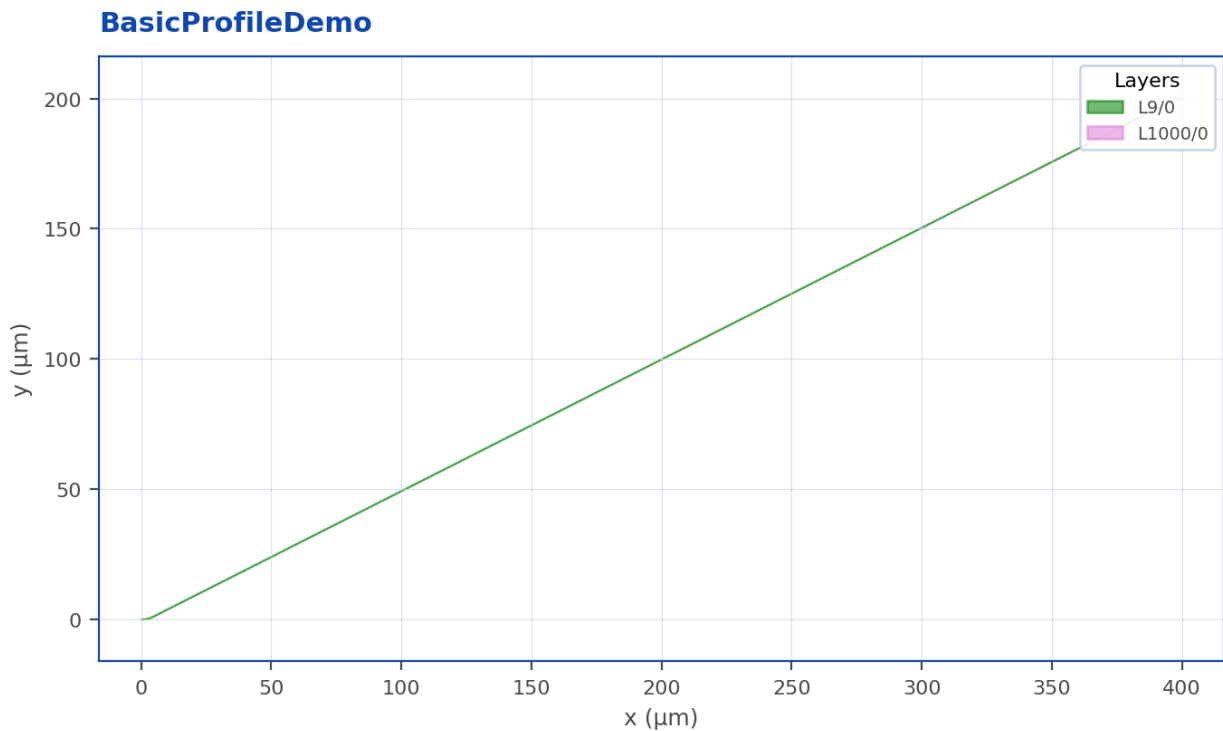


Figure 4.1.: Output of `03_route_profile_basic.py`.

That's a complete, usable profile. The route picks up its layer from `p.layer(...)`, its width from `p.port_width`, and its bend radius from the decorator's `radius=`.

4.3 Continuous strokes — `p.layer()`

```
p.layer(layer, *, width, offset=0.0)
```

Each call adds one parallel band. `offset` is perpendicular to the centerline (positive = left of travel direction). Common patterns:

```
# Symmetric cladding (claddings auto-center on the stroke).
p.layer(LAYER.OXID, width=p.port_width + 4.0)

# Slot-mode rail – second stroke on the SAME layer, offset.
p.layer(LAYER.SILC, width=0.2, offset=+0.6)
p.layer(LAYER.SILC, width=0.2, offset=-0.6)

# Dopant rail running parallel.
p.layer(LAYER.NDOP, width=2.0, offset=+1.5)
```

A working cladded profile:

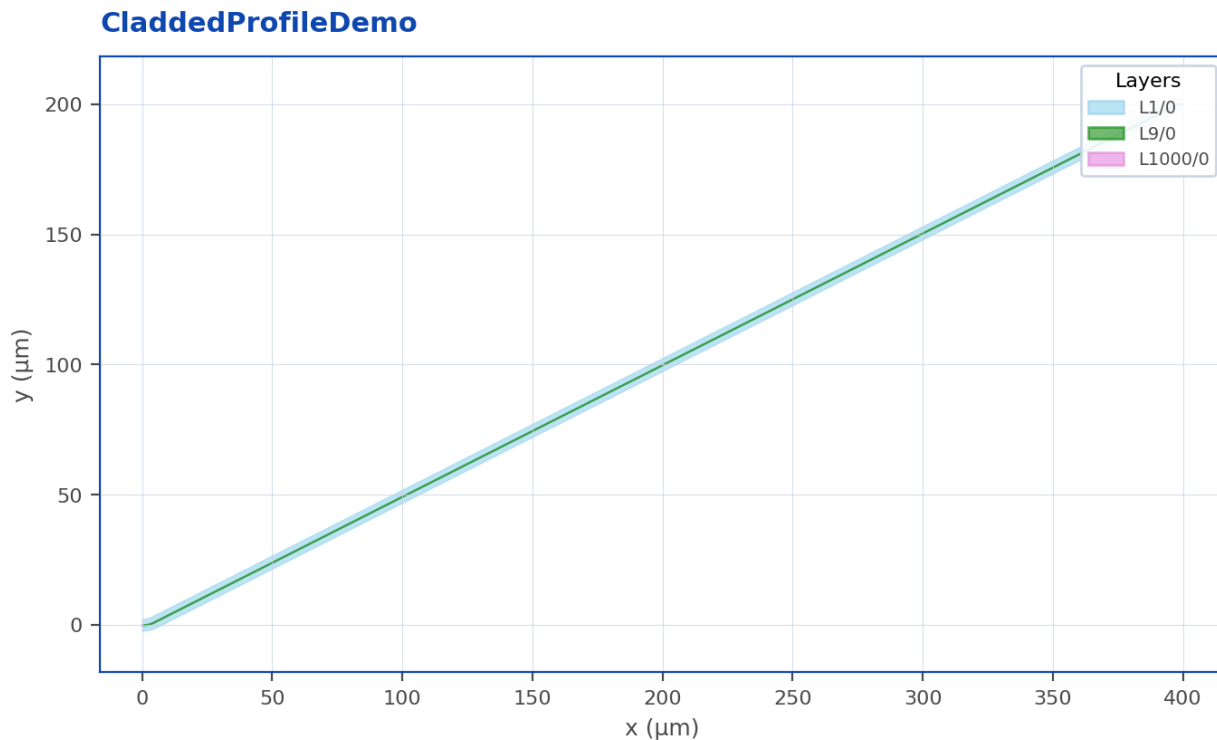


Figure 4.2.: Output of `04_route_profile_cladded.py` — silicon core with oxide cladding.

The router emits one `FlexRoute` per stroke. Bends share the same centerline, so widths and offsets stay consistent through corners.

4.4 Periodic features — p.tooth()

This is the part most other layout DSLs don't have a direct equivalent for. `p.tooth()` tiles a small PCell along the route at a fixed period:

```
p.tooth(cell, *, period, offset=0.0, phase=None, span_bends=True)
```

Arg	Purpose
cell	A @pcell PCell. Its local +x aligns with the path tangent.
period	Arc-length between consecutive teeth, in μm .
offset	Perpendicular offset from centerline (parallel side-cars).
phase	Arc-length of the first tooth. Default $\text{period}/2$ — center of slot 1.
span_bends	If False, skip teeth that would land inside a bend arc.

Use `p.tooth(...)` for:

- **Distributed Bragg gratings** — a wider rectangle every $\lambda/2n$.
- **Photonic-crystal slabs** — a small hole pattern repeated along the wave.
- **Periodic electrical contacts** — small metal stubs every $N \mu\text{m}$ for driven waveguides.
- **Mode-converter periodic claddings** — sub-wavelength teeth for SWG cross-sections.

A working grating profile:

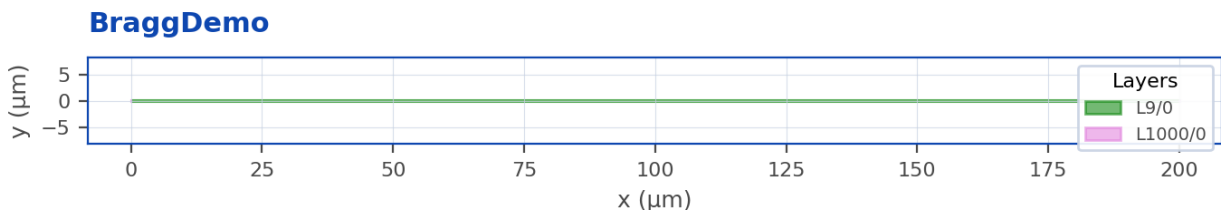


Figure 4.3.: Output of 05_route_profile_grating.py.

Warning

Teeth and bends. `span_bends=True` (default) tries to honor curvature — teeth re-orient with the path tangent through the arc — but for narrow-period gratings the tile-to-arc misfit becomes visible. For Bragg-period ($\lambda/2n \approx 320 \text{ nm}$) gratings, prefer `span_bends=False` and design your route to keep grating regions on straights. For coarse-period features ($\text{period} > 5 \times \text{bend radius}$) the default works.

Warning

Custom bend PCells. If a route uses a custom bend PCell (`bend=my_pcell`), teeth can't tile through the bend — the PCell renders its own geometry and there's no centerline to follow. The router emits a `ProfileWarning` and skips teeth that fall inside custom-bend segments.

4.5 Profile collections

After defining each profile, group them into a `RouteProfiles` namespace so end users have a single import:

```
RP = lm.RouteProfiles(
    silc_strip = silc_strip,
    silc_rib   = silc_rib,
    siln_strip = siln_strip,
    silc_bragg = silc_bragg,
)
```

End users then write `pdk.RP.silc_strip` — the same shape as `pdk.LAYER.SILC`. Iterating works too:

```
for name, profile in RP:
    print(name, profile)
```

4.6 Which knob the route reads

When several knobs disagree, this is the precedence (most-specific first):

1. **Section overrides** — `.style(...)`, `.radius(...)`, `.bend(...)`, `.width(...)` chained mid-route. Always win for the section they configure.
2. **Route(...) kwargs** — `radius=`, `bend=`, `profile=`, `width=` on the call. Override the port-attached profile.
3. **Port-attached profile** — `port_a.route_profile`. The default for the whole route.
4. **Decorator defaults** — `port_width`, `radius`, `radius_min` on the `@route_profile`.

So a PDK author shapes the *defaults* (steps 3-4); end users adjust the *overrides* (steps 1-2). When the defaults are good, end-user scripts are nearly kwarg-free.

4.7 Profile design patterns

A few patterns we've found pay back as a PDK grows:

One profile per layer × cross-section. Don't try to make a single profile parametric over layer choice — make `silc_strip` and `siln_strip` siblings. End-user scripts can switch via `.style(rp=...)` mid-route; PDK code stays linear.

Cladding widths follow the foundry rule, not the user's preference. If the foundry mandates a 4 μm OXID exclusion around all SILC, hard-code it: `p.layer(LAYER.OXID, width=p.port_width + 4.0)`. Don't expose it as a parameter — that just lets end users break the rule.

Tooth PCells live in the same module as the profile. A grating PCell that's only used by `silc_bragg` belongs in `profiles.py`, not `components.py`. End users don't reach for tooth cells directly; they get them transitively via the profile.

Name profiles by what they *do*, not what they're made of. `silc_strip` is fine. `silc_strip_v2_0p5um_with_ox` is not — the user shouldn't have to know about the cladding to pick a profile.

Listing 4.1 03_route_profile_basic.py

```

"""A minimal route profile – single core stroke.

The `@lm.route_profile` decorator turns a builder function into a
RouteProfile. Inside the body, call `p.layer(...)` once per continuous
stroke. The profile here defines just the silicon core – no claddings.
"""

import lumicron_api as lm

LAYER = lm.LayerTable(
    SILC = lm.Layer(9, 0, color="#008000", min_width=0.2, min_radius=5.0),
)

@lm.route_profile(port_width=0.5, radius=10.0, radius_min=5.0)
def silc_strip(p):
    """0.5 µm silicon core – minimal viable strip waveguide."""
    p.layer(LAYER.SILC, width=p.port_width)

# Optional: collect every profile this PDK exposes into one namespace,
# the way end-users will reference them as `pdk.RP.silc_strip`.
RP = lm.RouteProfiles(silc_strip=silc_strip)

@lm.pcell
def BasicProfileDemo():
    """Demo: route between two ports using the profile we just defined."""
    c = lm.CELL("BasicProfileDemo")
    c.add(lm.PORT(
        "in", position=(0, 0), direction=0,
        route_profile=silc_strip,
    ))
    c.add(lm.PORT(
        "out", position=(400, 200), direction=180,

        route_profile=silc_strip,
    ))
    c.Route(c.port["in"], c.port["out"])
    return c

if __name__ == "__main__":
    BasicProfileDemo().to_layout().to_gds("basic_profile_demo.gds")

```

Listing 4.2 04_route_profile_cladded.py

```

"""A cladded route profile – core + symmetric oxide cladding.

`p.layer()` can be called any number of times. Each call adds one
parallel stroke to the cross-section. Use it for claddings, exclusion
zones, slot-mode rails, or any sibling band that should run alongside
the core.
"""
import lumicron_api as lm

LAYER = lm.LayerTable(
    SILC = lm.Layer(9, 0, color="#008000", min_width=0.2, min_radius=5.0),
    OXID = lm.Layer(1, 0, color="#87ceeb"),
)

@lm.route_profile(port_width=0.5, radius=10.0, radius_min=5.0)
def silc_cladded(p):
    """Silicon core with a 4 μm oxide cladding on either side."""
    p.layer(LAYER.SILC, width=p.port_width)
    p.layer(LAYER.OXID, width=p.port_width + 4.0)

@lm.pcell
def CladdedProfileDemo():
    c = lm.CELL("CladdedProfileDemo")
    c.add(lm.PORT(
        "in", position=(0, 0), direction=0,
        route_profile=silc_cladded,
    ))
    c.add(lm.PORT(
        "out", position=(400, 200), direction=180,
        route_profile=silc_cladded,
    ))
    c.Route(c.port["in"], c.port["out"])
    return c

if __name__ == "__main__":
    CladdedProfileDemo().to_layout().to_gds("cladded_profile_demo.gds")

```

Listing 4.3 05_route_profile_grating.py

```

"""A periodic route profile – distributed Bragg grating.

`p.tooth(cell, period=...)` tiles a small PCell along the route. Use it
for grating teeth, periodic claddings, photonic-crystal slabs, or
periodic electrical contacts. The cell's local +x aligns with the path
tangent, so a single 1-period unit cell is enough.
"""

import lumicron_api as lm

LAYER = lm.LayerTable(
    SILC = lm.Layer(9, 0, color="#008000", min_width=0.2, min_radius=5.0),
)

@lm.pcell
def BraggTooth(width: float = 1.5, length: float = 0.16):
    """One tooth of a distributed Bragg grating – a wider rectangle."""
    c = lm.CELL("BraggTooth")
    rect = lm.Rectangle(x_dim=length, y_dim=width, layer=LAYER.SILC)
    h = c.add(rect)
    c.Place(h).at((0, 0))
    return c

@lm.route_profile(port_width=0.5, radius=20.0, radius_min=10.0)
def silc_bragg(p):
    """Silicon strip with a 320 nm-period Bragg grating overlaid."""
    p.layer(LAYER.SILC, width=p.port_width)
    # Period = 320 nm; tile only on straight segments (span_bends=False)
    # since the wide teeth would distort an arc.
    p.tooth(BraggTooth(width=1.5, length=0.16),
            period=0.32, span_bends=False)

@lm.pcell
def BraggDemo():
    c = lm.CELL("BraggDemo")
    c.add(lm.PORT("in", position=(0, 0), direction=0,
                 route_profile=silc_bragg))
    c.add(lm.PORT("out", position=(200, 0), direction=180,
                 route_profile=silc_bragg))
    c.Route(c.port["in"], c.port["out"])
    return c

if __name__ == "__main__":
    BraggDemo().to_layout().to_gds("bragg_demo.gds")

```

CHAPTER 5

Components

Components are the named building blocks your PDK exposes — pads, edge couplers, MMIs, modulators, photodiodes. Each is a `@pcell`-decorated function that returns a `CELL`. End users instantiate them like functions; the decorator handles cell naming, parameter hashing, and re-use.

5.1 A simple component

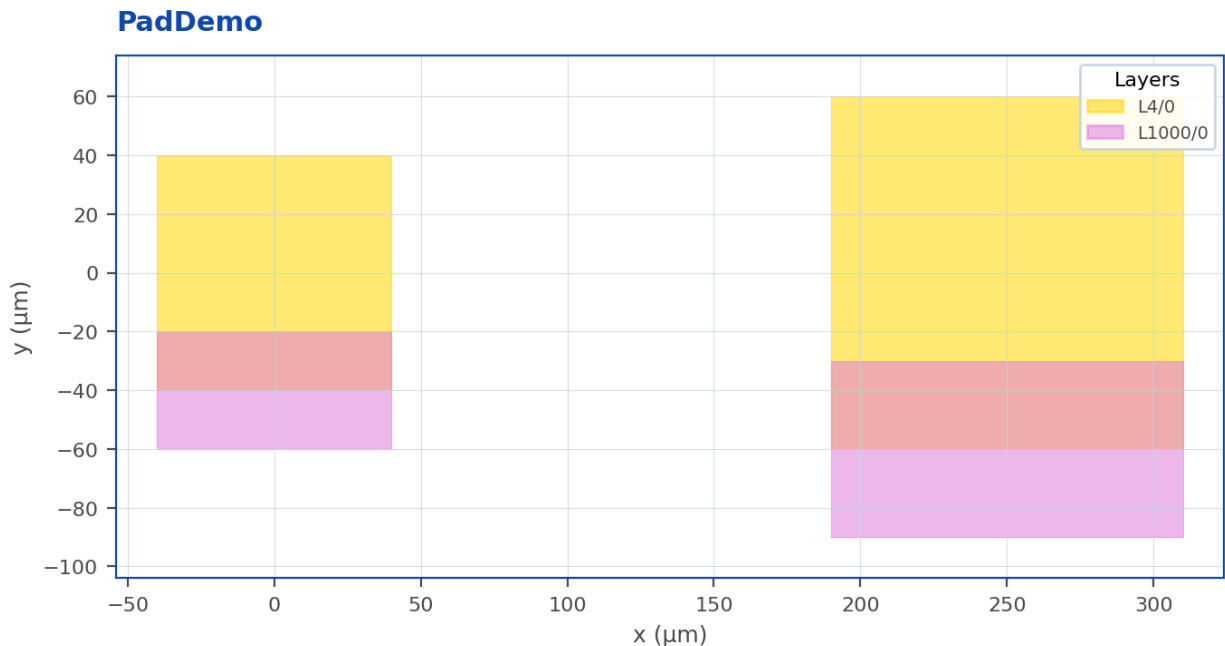


Figure 5.1.: Output of `06_simple_component.py` — two pads of different sizes.

What `@pcell` is doing for you:

- **Auto-naming.** Each unique combination of non-default parameters yields a distinct cell name in the GDS. Two `Pad(size=80)` calls share one cell definition; `Pad(size=120)` gets its own.
- **Parameter hashing.** Internally the decorator hashes the kwargs into the cell name, so name collisions across many parameter combinations are vanishingly unlikely.
- **Caching.** Calling `Pad(size=80)` twice returns the same `CELL` object — the second call is essentially free.

This is the same shape as IPKISS PCells or KFactory cells — if you know one, you know this.

5.2 Publishing ports with route profiles attached

This is the workflow that makes a polished PDK. Every port your component publishes carries a `route_profile=` so end-user routes off it inherit the cross-section automatically.

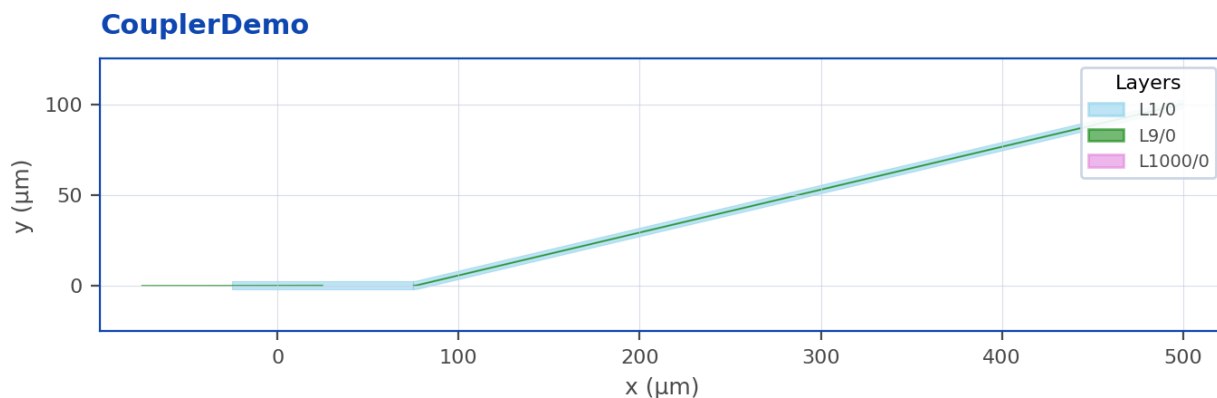


Figure 5.2.: Output of `07_component_with_profile.py` — coupler plus auto-routed exit.

The relevant lines:

```
c.add(lm.PORT("o1", position=(length, 0), direction=0,
              width=body_w, route_profile=silc_strip))
```

Then in the end-user demo at the bottom:

```
chip.Route(cpl.port["o1"], chip.port["out"])
```

No `radius=`, no `profile=` — the route inherits both from the port. The user's script reads like a circuit diagram instead of a parameter dump.

5.3 What “good ports” look like

Three rules of thumb for component port design:

Use the same port-name convention everywhere. `i1/o1` for two-port elements, `i1`, `o1`, `o2` for splitters, `o_tip` for edge-coupler facet ports. Once an end user has learned your convention, they shouldn’t have to consult docs to route between any two of your components.

Direction matters. Optical ports point *outward* from the cell — `direction=0` (East) for a port on the right edge, `180` (West) for the left, `90` for the top, `270` for the bottom. The router uses port direction to pick Manhattan vs direct routing.

Always attach a route profile. If a port is on a layer your PDK has no profile for, add a profile for it. The cost of an extra profile is one decorated function; the payoff is end-user scripts that route through your component without thinking.

5.4 Synthesizing geometry

Inside the component body, you have the full `lumicron_api` toolkit:

Tool	When to use it
<code>lm.Rectangle(x_dim, y_dim, layer)</code>	Square / rectangular regions, pads, body slabs.
<code>lm.Polygon(vertices=[...], layer)</code>	Tapers, spline-like profiles, anything non-rectangular.
<code>lm.Ring(outer_radius, width, layer)</code>	Ring resonators, annular regions.
<code>lm.Circle(radius, layer)</code>	Filled circles — alignment marks, photonic-crystal holes.
<code>lm.ARRAY(child, count, pitch)</code>	Periodic features — via arrays, contact arrays.
<code>lm.Waveguide(points, layer, width, ...)</code>	Tapered curvy paths (S-bends in components, spirals).
<code>c.add(child_cell) + c.Place(...)</code>	Hierarchical sub-components.

5.5 Promoting child ports

When a component contains sub-cells whose ports you want to expose as the component’s own:

```
@lm.pcell
def MZI(length: float = 200, width: float = 0.5):
    c = lm.CELL("MZI")
    splitter = c.add(YSplitter(width=width))
```

```

combiner = c.add(YSplitter(width=width))
c.Place(splitter).at((0, 0))
c.Place(combiner).at((length + 60, 0)).rotate_by(180)

# Re-publish splitter's input as MZI's input.
splitter.promote(ports=["i1"])
# Re-publish combiner's input as MZI's output, with a renamed key.
combiner.promote(ports={"i1": "o1"})

return c

```

`handle.promote()` accepts a list of names, a rename dict, or `None` (everything). An optional `prefix=` namespaces them when multiple children expose ports of the same name.

5.6 Component design patterns

Sensible defaults beat exhaustive parameters. Pick sensible foundry-typical defaults for everything; expose the kwargs that real users will actually want to vary. A modulator with 30 parameters is hard to use; one with 5 parameters and good defaults is easy.

Match the foundry datasheet's parameter names. If the foundry's PDK calls it `arm_length`, name your kwarg `arm_length`. End users cross-reference the datasheet while writing scripts.

Validate inputs early. Raise `ValueError` with a useful message when a parameter is out of range. Better to fail at component construction than at GDS-write time.

Set the cell name from a static string + the decorator handles the rest. The first argument to `lm.CELL("ComponentName")` is the *base* name; `@pcell` appends a hash of the parameters. Don't try to compose the full name yourself.

Tip

A useful trick when porting an existing PDK: write each component's tests *first*, expressing what end-user scripts should look like (`Pad(size=80).port["p"]`, `MZI(length=200).port["i1"]`). Then implement the components to make the tests pass. The result is a PDK whose API was designed from the user's seat.

Listing 5.1 06_simple_component.py

```

"""A simple PDK component with @pcell.

The `@pcell` decorator turns a function into a parametrized cell
factory. Each unique combination of non-default arguments yields a
distinct cell name automatically – distinct calls don't collide in GDS,
identical calls share the same cell definition under the hood.
"""

import lumicron_api as lm

LAYER = lm.LayerTable(
    MTL1 = lm.Layer(4, 0, color="#ffd700", min_width=1.0, min_spacing=1.0),
)

@lm.pcell
def Pad(size: float = 80.0, layer: lm.LayerSpec = LAYER.MTL1):
    """A square electrical bond pad with one electrical port."""
    c = lm.CELL("Pad")

    body = lm.Rectangle(x_dim=size, y_dim=size, layer=layer)
    h = c.add(body)
    c.Place(h).at((0, 0))

    # Publish a port at the south edge so routes can land on it.
    c.add(lm.PORT(
        "p", position=(0, -size / 2), direction=270,
        width=size, layer=layer, port_type="electrical",
    ))
    return c

@lm.pcell
def PadDemo():
    """Two pads side-by-side at different sizes – different cell names auto-derived."""
    c = lm.CELL("PadDemo")
    p1 = c.add(Pad(size=80))
    p2 = c.add(Pad(size=120))
    c.Place(p1).at((0, 0))
    c.Place(p2).at((250, 0))
    return c

if __name__ == "__main__":
    PadDemo().to_layout().to_gds("pad_demo.gds")

```

Listing 5.2 07_component_with_profile.py

```

"""A component publishing ports with an attached route profile.

The trick that makes a well-curated PDK feel effortless: every
component publishes ports that already know their cross-section. End
users call ``c.Route(a.port["o1"], b.port["i1"])`` with no kwargs and
the route automatically stamps the right layers, claddings, and
periodic features.
"""

import lumicron_api as lm

LAYER = lm.LayerTable(
    SILC = lm.Layer(9, 0, color="#008000", min_width=0.2, min_radius=5.0),
    OXID = lm.Layer(1, 0, color="#87ceeb"),
)

@lm.route_profile(port_width=0.5, radius=10.0, radius_min=5.0)
def silc_strip(p):
    p.layer(LAYER.SILC, width=p.port_width)
    p.layer(LAYER.OXID, width=p.port_width + 4.0)

@lm.pcell
def EdgeCouplerSi(length: float = 100.0, tip_width: float = 0.1):
    """A linear taper from a 0.1  $\mu\text{m}$  tip to a 0.5  $\mu\text{m}$  body, with an
    OXID cladding region around it. The output port carries
    `silc_strip` so any route off it inherits the profile."""
    c = lm.CELL("EdgeCouplerSi")

    # The taper polygon: tip  $\rightarrow$  body, 4 vertices.
    body_w = 0.5
    taper = lm.Polygon(
        vertices=[
            (0, -tip_width / 2),
            (length, -body_w / 2),
            (length, body_w / 2),
            (0, tip_width / 2),
        ],
        layer=LAYER.SILC,
    )
    c.add(taper)
    c.Place(taper).at((0, 0))

    # OXID cladding rectangle hugging the taper.
    clad = lm.Rectangle(x_dim=length, y_dim=body_w + 4, layer=LAYER.OXID)
    h = c.add(clad)
    c.Place(h).using("SW").at((0, -(body_w + 4) / 2))

    # Tip port (incoming from chip edge) and body port (publishes profile).
    c.add(lm.PORT("o_tip", position=(0, 0), direction=180,
                  width=tip_width, layer=LAYER.SILC))
    c.add(lm.PORT("o1", position=(length, 0), direction=0,
                  width=body_w, route_profile=silc_strip))

    return c

```

CHAPTER 6

Layer transitions

When a route's `port_a` and `port_b` live on different layers, the router checks a `TransitionRegistry`. If the layer pair has a registered factory, the router automatically inserts the resulting PCell — typically an inverse-taper bridge — at a configurable point along the route.

If the pair isn't registered, the route raises. So PDK authors decide which cross-layer routes are sanctioned, and end users get either a clean transition or an actionable error.

6.1 Registering a transition

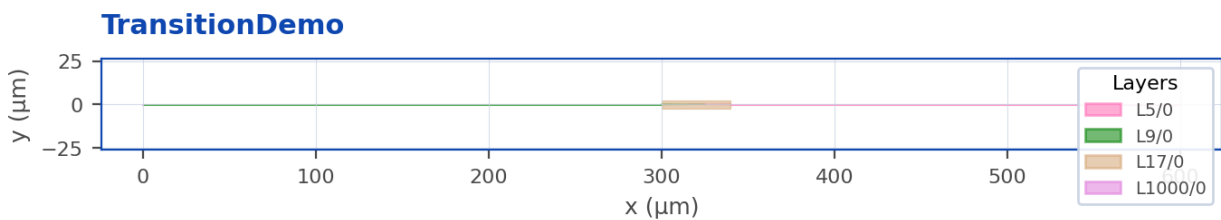


Figure 6.1.: Output of `08_transition_pcell.py` — auto-inserted SiLC SiLN elevator.

Two registries appear:

```
TRANSITIONS = lm.TransitionRegistry() # PDK-local
TRANSITIONS.register(LAYER.SILC, LAYER.SILN, factory)

lm.default_transitions().register(LAYER.SILC, LAYER.SILN, factory) # process-wide
```

- The **PDK-local** registry is what `pdk.TRANSITIONS` exposes for users who want to pass it explicitly: `chip.Route(a, b, transitions=pdk.TRANSITIONS)`.

- The **process-wide** registry is read when no explicit `transitions=` is passed. Registering into it from your PDK module — typically inside the `transitions.py` module body — means importing the PDK is enough to wire up auto-transitions everywhere.

6.2 The factory contract

A transition entry maps `(layer_a, layer_b)` to a **factory function**, not a pre-built cell:

```
def silc_siln_elevator():  
    return lm.PhotonicElevator(...)
```

The router calls the factory each time a route needs the transition. Why a function rather than a cached cell:

- The router can pass position-derived kwargs in the future (e.g. mid-bend transition would need a curved entry).
- Multiple routes through the same transition each get their own SRef of a fresh PCell instance, so per-instance overrides (cladding margin variants) stay clean.
- It side-steps Python's import-time evaluation: the factory body can reference layers and profiles defined in any order.

In practice the factory is a one-liner that constructs and returns a PCell. The PDK-shipped `lm.PhotonicElevator` is the most common building block — two inverse tapers stacked with an overlap region — but any CELL with two ports works.

6.3 What a transition cell needs

The router's contract with a transition cell is minimal:

- **One input port named `i1`** on `layer_a` (matching the route's incoming layer).
- **One output port named `o1`** on `layer_b` (matching the route's outgoing layer).
- Both ports should carry a `route_profile=` so the route's continuation inherits the right cross-section.

Anything else — internal taper geometry, cladding patterns, doping rails — is up to the cell.

6.4 Where transitions appear in a route

The router places the transition at a position controlled by the route call:

```
chip.Route(a, b, transition_offset=300)
```

`transition_offset=N` means “place the transition $N \mu\text{m}$ in from `port_a`”. If you don't pass it, the router defaults to a reasonable midpoint.

For section-level layer changes (`.style(rp=siln_strip)` mid-route), the router inserts a transition at the section boundary using the same registry. So one registration covers all uses — full-route changes and mid-route changes both.

6.5 When the registry doesn't have an entry

```
RouteError: no transition registered for layer pair SILC → MTL1
→ Register a factory with TransitionRegistry.register(layer_a, layer_b, factory),
or stay on a single layer.
```

This is intentional. Photonic ↔ electrical, or photonic ↔ photonic across very different stacks (silicon ↔ III-V), aren't well-defined transitions — they need a designed PCell that the foundry has characterized. Forcing PDK authors to register transitions surfaces those design decisions instead of letting the router silently butt-join.

Tip

Register transitions in both directions if your process supports them. `register(SILC, SILN, ...)` does NOT auto-register `SILN, SILC, ...`. The PhotonicElevator is symmetric, so a single factory works for both directions — call `TRANSITIONS.register` twice with swapped layers.

6.6 What ships with the API

Two transition primitives are bundled:

Primitive	What it does
<code>lm.PhotonicElevator</code>	Two inverse tapers stacked vertically with an overlap. Standard photonic interlayer transition.
<code>lm.TransitionRegistry</code>	The registry container itself.

Custom transition PCells (mode-matched bridges, sub-wavelength couplers, polarization rotators) are PDK-specific. They're just `@pcell` cells with `i1/o1` ports — write them like any other component and register the factory.

Listing 6.1 08_transition_pcell.py

```

"""Registering a layer transition.

When a route's port_a and port_b live on different layers, the router
consults a `TransitionRegistry`. If the layer pair has a registered
factory, the router drops in the resulting PCell automatically. Here
we wire SILC ↔ SILN with the built-in `PhotonicElevator`.
"""

import lumicron_api as lm

LAYER = lm.LayerTable(
    SILC = lm.Layer(9, 0, color="#008000", min_width=0.2, min_radius=5.0),
    SILN = lm.Layer(5, 0, color="#ff69b4", min_width=0.4, min_radius=10.0),
    OXOP = lm.Layer(17, 0, color="#d4a574"),
)

@lm.route_profile(port_width=0.5, radius=10.0, radius_min=5.0)
def silc_strip(p):
    p.layer(LAYER.SILC, width=p.port_width)

@lm.route_profile(port_width=0.5, radius=10.0, radius_min=5.0)
def siln_strip(p):
    p.layer(LAYER.SILN, width=p.port_width)

def silc_siln_elevator():
    """Factory: returns a fresh SILC ↔ SILN elevator each call.

    The router calls this every time a route needs the transition,
    so the registry stores the function rather than a cached cell.
    """
    return lm.PhotonicElevator(
        layer_a=LAYER.SILC, layer_b=LAYER.SILN,
        width_a=0.5, width_b=0.5,
        tip_a=0.08, tip_b=0.08,
        taper_length_a=25.0, taper_length_b=25.0,
        overlap=10.0,
        cladding_layer=LAYER.OXOP, cladding_margin=2.0,
        route_profile_a=silc_strip,
        route_profile_b=siln_strip,
    )

# Register both into a PDK-local registry and into the process-wide
# default registry, so end-user scripts get auto-transitions just by

# importing the PDK.
TRANSITIONS = lm.TransitionRegistry()
TRANSITIONS.register(LAYER.SILC, LAYER.SILN, silc_siln_elevator)
lm.default_transitions().register(LAYER.SILC, LAYER.SILN, silc_siln_elevator)

@lm.pcell
def TransitionDemo():

```

Part III.

Packaging & distribution

CHAPTER 7

Packaging — the .lumpdk format

A `.lumpdk` is a zip archive containing a manifest and the Python source for a PDK. The Lumicron Mac app installs it by extracting into `~/Library/Application Support/Lumicron/UserPDKs/`, where it joins the import path as `lumicron_api.pdks.<your_pdk>`.

You don't need a `.lumpdk` to ship a PDK on PyPI — `pip install your_pdk` works fine for users who script outside the app. But the app's PDK panel installs are `.lumpdk`-only, so if you want one-click installation, package both.

7.1 Anatomy

```
your_pdk.lumpdk          # zip archive
├─ manifest.json         # required: name, version, root module, layer map
├─ source/               # required: Python package source tree
│   └─ your_pdk/
│       ├── __init__.py
│       ├── all.py
│       ├── layers.py
│       ├── profiles.py
│       ├── components.py
│       └─ transitions.py
```

`source/<your_pdk>/` is what gets symlinked into the app's `UserPDKs/` directory. Its contents are an ordinary Python package — what you'd publish to PyPI minus the wheel boilerplate.

7.2 The manifest

`manifest.json` is the metadata index. Minimum:

```
{
  "name": "TinyPhot",
  "module": "tinyphot",
  "version": "0.1.0",
  "lumicron_api_required": ">=0.4",
  "description": "Compact silicon photonics PDK for tutorials.",
  "publisher": "Elyon Semiconductor",
  "license": "MIT",
  "layers": [
    {"name": "SILC", "layer": 9, "datatype": 0, "color": "#008000"},
    {"name": "SILN", "layer": 5, "datatype": 0, "color": "#ff69b4"},
    {"name": "OXID", "layer": 1, "datatype": 0, "color": "#87ceeb"},
    {"name": "MTL1", "layer": 4, "datatype": 0, "color": "#ffd700"}
  ]
}
```

Field reference is in Appendix A. Two fields worth highlighting:

- **module** is the import path — import lumicron_api.pdks.<module>.all as pdk. It must match the directory name under source/.
- **layers** is what the Mac app reads to populate the layers panel without importing your Python source. Generate it from LAYER.to_manifest() so the Python truth and the JSON stay in sync.

7.3 Building a .lumpdk

A minimal build script:

```
import json
import zipfile
from pathlib import Path

import your_pdk
from your_pdk.layers import LAYER

PDK_NAME = "TinyPhot"
MODULE = "tinyphot"
VERSION = "0.1.0"

manifest = {
    "name": PDK_NAME,
    "module": MODULE,
    "version": VERSION,
    "lumicron_api_required": ">=0.4",
    "description": "Compact silicon photonics PDK.",
    "publisher": "Elyon Semiconductor",
    "license": "MIT",
```

```

    "layers": LAYER.to_manifest(),
}

src_dir = Path(your_pdk.__file__).parent
out = Path(f"{PDK_NAME.lower()}-{VERSION}.lumpdk")

with zipfile.ZipFile(out, "w", zipfile.ZIP_DEFLATED) as z:
    z.writestr("manifest.json", json.dumps(manifest, indent=2))
    for path in src_dir.rglob("*.py"):
        rel = path.relative_to(src_dir.parent)
        z.write(path, f"source/{rel}")

print(f"wrote {out}")

```

Run it from the directory where `your_pdk` is importable; the result is a self-contained archive ready to drop into the Mac app.

Warning

Compression. Use `zipfile.ZIP_DEFLATED` (or `ZIP_STORED` for faster install) — both work. The Lumicron app's .lumpdk reader supports both. Avoid `ZIP_LZMA` and `ZIP_BZIP2` — Apple's Compression framework doesn't decompress them out of the box and the install will fail silently on iCloud-stored files.

7.4 Where it lands after install

When a user double-clicks `tinypshot-0.1.0.lumpdk` (or installs via Settings ▢ PDKs):

1. The app verifies the manifest against the bundle contents.
2. The `source/tinypshot/` tree is extracted into `~/Library/Application Support/Lumicron/UserPDKs/tinypshot`.
3. The app appends `~/Library/Application Support/Lumicron/UserPDKs/` to `lumicron_api.pdks.__path__` at startup, so user scripts can import `lumicron_api.pdks.tinypshot.all` as `pdk`.
4. The layer map from the manifest is registered in `TechnologyStore` so the layers panel shows your colors and names.

For users who'd rather install via `pip` (e.g. CI environments without the Mac app), publishing to PyPI in parallel is supported — see Chapter 8.

7.5 Versioning the bundle

Two versions live in the manifest:

- **version** — your PDK’s own semver. Bump on changes to layers, profiles, or components. End-user scripts that depend on a specific cross-section should pin this.
- **lumicron_api_required** — the minimum `lumicron_api` your PDK needs. Use a soft floor (≥ 0.4) unless you’ve adopted a feature only present in a specific release.

The Mac app refuses to install a `.lumpdk` whose `lumicron_api_required` is newer than the bundled `lumicron_api` in the app — keeps users from getting cryptic `ImportErrors` when they upgrade ahead of the app.

7.6 What can go wrong

Symptom	Likely cause
Install reports “no Python source”	Bundle is missing <code>source/<module>/</code> directory.
Import works but layers don’t render colored	<code>manifest.json</code> <code>layers[]</code> empty or layer names don’t match Python <code>LayerTable</code> keys.
<code>lumicron_api</code> version too old	Bundled <code>lumicron_api</code> predates a feature your PDK uses. Bump the app or relax <code>lumicron_api_required</code> .
User reports <code>ImportError: no module named X</code>	A relative import inside your PDK references a module not bundled in <code>source/</code> .

When testing, use `python -c "import lumicron_api.pdks.<your_pdk>.all"` from a clean shell — if that works, the bundle works.

CHAPTER 8

Publishing

Three audiences, three distribution channels:

- **Mac app users** — install via the in-app PDK panel from a `.lumpdk` file you publish.
- **Python scripters** — `pip install your_pdk` from PyPI or a private index.
- **Internal teams** — a shared NFS / git checkout, or a private PyPI mirror.

You don't have to pick one. A typical foundry PDK ships all three from the same source tree.

8.1 Versioning

Use [Semantic Versioning](#). What “breaking” means for a PDK:

Change	Bump
New component / new profile / new layer	minor
New design rule on an existing layer	minor
Renamed component or layer	major
Tightened a <code>min_*</code> rule	major
Changed a profile's strokes (anything visible in GDS)	major
Changed default kwargs of a component	major
Bug fix that doesn't move geometry by $\geq \epsilon$	patch

End-user scripts pin `your_pdk == 1.2.*` for a stable cross-section; pinning `>= 1.2` accepts new components but not breaking changes.

8.2 Publishing to PyPI

Your PDK is an ordinary Python package — `pyproject.toml`, a wheel, `pip install`. The convention for placement:

```
[project]
name = "lumicron-pdk-tinyphot"
version = "0.1.0"
requires-python = ">=3.12"
dependencies = ["lumicron_api>=0.4"]

[tool.setuptools.packages.find]
where = ["src"]
include = ["lumicron_api.pdks.*"]
namespace_packages = true
```

Two non-obvious bits:

- **Namespace package.** `lumicron_api.pdks` is intentionally a namespace package — multiple PDKs install side-by-side under it. Don't ship a `lumicron_api/__init__.py` or `lumicron_api/pdks/__init__.py` from your wheel. `setuptools.packages.find` with `namespace_packages = true` handles it.
- **Source layout.** Put your code at `src/lumicron_api/pdks/<your_pdk>/.` The PyPI wheel and the `.lumpdk` bundle both consume the same directory.

Distribution name conventions:

- `lumicron-pdk-<your_pdk>` — recommended. Distinct on PyPI, telegraphs what it is.
- `<your_pdk>-pdk-lumicron` — if your PDK has a strong existing brand (e.g. `sky130`).

8.3 Publishing a `.lumpdk`

Two options:

- **Direct download** — host `tinyphot-1.2.0.lumpdk` on your website / S3 / GitHub releases. Users download and double-click.
- **Marketplace** — submit to the in-app marketplace (forthcoming). The app fetches the latest version directly. Useful for paid or subscription PDKs.

For now (early 2026), direct download is the dominant channel. The marketplace launches alongside the v1 paid release of Lumicron Mac.

8.4 What to put in your README

The conventional shape works:

1. **One-line description.** What process, what's special.
2. **Install** — `pip install ...` and `.lumpdk` link.
3. **Quick start** — 5-10 line script that imports the PDK and routes between two of its components. Make sure it actually runs.
4. **Layer table** — the output of `LAYER.print_rules()`, copy-pasted.
5. **Component list** — names + 1-line descriptions.
6. **Foundry / process notes** — substrate, BOX thickness, key min rules.
7. **License + foundry agreement notes** — if your PDK includes confidential geometry, say so.

End users find a foundry PDK by Googling — make the README's first line match what they'll search for.

8.5 CI / release pipeline

Recommended steps for a tagged release:

1. **Run the test suite.** Every component instantiates without raising, every profile renders a non-empty FlexRoute, every cross-layer route uses the registered transition.
2. **Run end-to-end script tests.** A handful of scripts that import your PDK, build a small chip, and verify the resulting GDS has the expected layers and reasonable polygon counts. Catches regressions in routing / transitions that unit tests miss.
3. **Build the wheel** — `python -m build`.
4. **Build the .lumpdk** — the Chapter 7 script.
5. **Publish to PyPI** — `twine upload dist/*.whl`.
6. **Attach the .lumpdk to the GitHub release.**

The whole pipeline takes ~30 seconds for a small PDK. Wire it up early — it's much harder to retrofit.

8.6 Marketing your PDK

If you're publishing to a wider audience:

Show, don't list. A short video or screen recording of a script that imports your PDK and routes between two non-trivial components is worth more than a feature list.

Be specific about what's *not* covered. A PDK that says “production-ready for [process]” but secretly omits half the device library will burn its first user. Be honest about scope.

Pin a tested `lumicron_api` version. End users running an older `lumicron_api` shouldn't see cryptic errors — `lumicron_api_required` in the manifest catches this with a clear message.

Part IV.

Cookbook

CHAPTER 9

Cookbook — a minimal PDK from scratch

Everything in one file. Layers, profiles, components, transitions. In a real distribution each chunk lives in its own module; here it's collapsed so you can see the whole thing at once.

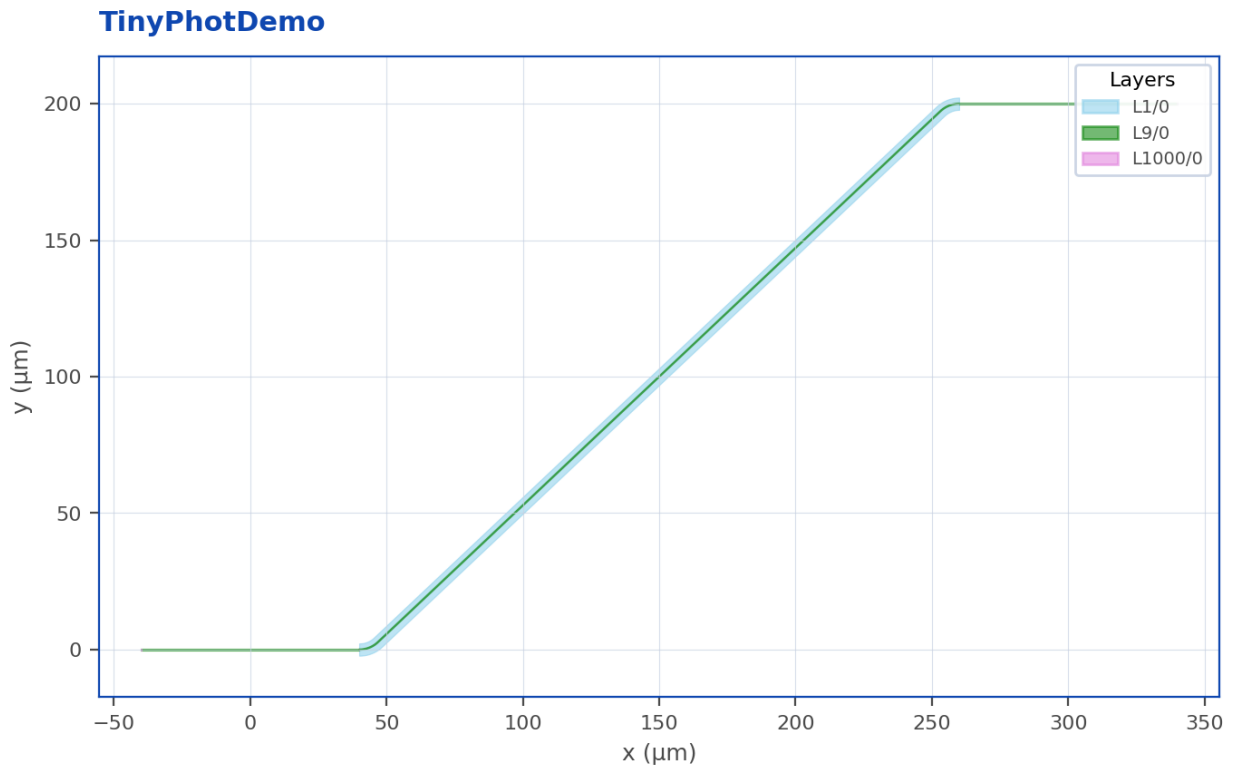


Figure 9.1.: Output of `09_full_minimal_pdk.py` — two waveguide stubs joined by an auto-route.

9.1 Splitting it into a package

To turn this single file into the distributable layout from Chapter 1, copy each section into its own module:

tinyphot/layers.py LAYER and STACK definitions. Imports only lumicron_api.

tinyphot/profiles.py silc_strip, siln_strip, RP. Imports LAYER from .layers.

tinyphot/components.py Pad, Waveguide. Imports LAYER and the route profiles.

tinyphot/transitions.py silc_siln_elevator(), TRANSITIONS, the default_transitions().register(...) call. Imports LAYER and RP.

tinyphot/__init__.py Re-exports the four names.

tinyphot/all.py

```
from tinyphot.layers import LAYER, STACK
from tinyphot.profiles import RP
from tinyphot.transitions import TRANSITIONS
from tinyphot.components import * # noqa
```

That's the conventional shape end users see. The same source tree ships in your PyPI wheel and inside the source/tinyphot/ directory of your .lumpdk.

9.2 What this minimal PDK is missing

A production-grade PDK has more than this:

- **More components.** Edge couplers, MMIs, modulators, photodiodes, ring resonators with thermal heaters, contact arrays.
- **More profiles.** A rib profile, a slot-mode profile, a periodic-cladding sub-wavelength profile, an electrical profile.
- **More transitions.** SILC → MTL1 ohmic contact, SILN → III-V wafer bond.
- **A proper LayerStack with Sellmeier coefficients.** This minimal stack would not power a usable mode solver.
- **Test scripts.** test_pad.py, test_components_smoke.py.
- **Documentation.** README with examples; per-component .rules() strings.

You add these as you go — start with the four-section shape and grow the package.

9.3 Verifying it works

Drop the file at `tinyphot/__init__.py` (alongside `lumicron_api`'s package directory in your environment, or anywhere on `PYTHONPATH`), then:

```
python -c "  
import lumicron_api as lm  
import tinyphot as pdk  
  
chip = lm.CELL('Smoke')  
wg = chip.add(pdk.Waveguide(length=200))  
chip.Place(wg).at((0, 0))  
chip.to_layout().to_gds('smoke.gds')  
"
```

If `smoke.gds` lands and contains a SILC strip 200 μm long, your PDK is wired up. From there, it's a matter of growing the component library and refining the rules.

Listing 9.1 09_full_minimal_pdk.py

```

"""A complete, single-file minimal PDK.

In a real distribution, layers / profiles / components / transitions
each live in their own module. For learning purposes everything is
collapsed into one file here. The end-user import surface is the same:

    import lumicron_api.pdks.tinyphot.all as pdk
    pdk.LAYER.SILC
    pdk.RP.silc_strip
    pdk.Pad(size=80)
"""

import lumicron_api as lm

# — Layers —————
LAYER = lm.LayerTable(
    SILC = lm.Layer(9, 0, color="#008000", min_width=0.2, min_radius=5.0),
    SILN = lm.Layer(5, 0, color="#ff69b4", min_width=0.4, min_radius=10.0),
    OXID = lm.Layer(1, 0, color="#87ceeb"),
    MTL1 = lm.Layer(4, 0, color="#ffd700", min_width=1.0, min_spacing=1.0),
)

# — Route profiles —————
@lm.route_profile(port_width=0.5, radius=10.0, radius_min=5.0)
def silc_strip(p):
    p.layer(LAYER.SILC, width=p.port_width)
    p.layer(LAYER.OXID, width=p.port_width + 4.0)

@lm.route_profile(port_width=0.5, radius=10.0, radius_min=5.0)
def siln_strip(p):
    p.layer(LAYER.SILN, width=p.port_width)
    p.layer(LAYER.OXID, width=p.port_width + 4.0)

RP = lm.RouteProfiles(silc_strip=silc_strip, siln_strip=siln_strip)

# — Components —————
@lm.pcell
def Pad(size: float = 80.0):
    c = lm.CELL("Pad")
    body = lm.Rectangle(x_dim=size, y_dim=size, layer=LAYER.MTL1)
    h = c.add(body)
    c.Place(h).at((0, 0))
    c.add(lm.PORT("p", position=(0, -size / 2), direction=270,
                  width=size, layer=LAYER.MTL1, port_type="electrical"))
    return c

@lm.pcell
def Waveguide(length: float = 100.0):
    """A straight waveguide stub – useful for grating couplers, taps."""
    c = lm.CELL("Waveguide")
    rect = lm.Rectangle(x_dim=length, y_dim=0.5, layer=LAYER.SILC)
    h = c.add(rect)
    c.Place(h).using("SW").at((0, -0.25))

```

Manifest schema

Every `.lumpdk` archive contains a `manifest.json` at its root. This is the full field reference.

Required fields

Field	Type	Description
<code>name</code>	string	Display name shown in the Lumicron Mac app's PDK panel.
<code>module</code>	string	Python import path under <code>lumicron_api.pdks.<module></code> . Must match the directory name under <code>source/</code> . Lowercase, snake_case.
<code>version</code>	string	Semver. End users pin against this.
<code>lumicron_api_required</code>	string	PEP-508 version specifier. App refuses to install if the bundled <code>lumicron_api</code> is older. Example: <code>">=0.4,<0.6"</code> .

Recommended fields

Field	Type	Description
<code>description</code>	string	One-line summary. Shown under the name in the PDK panel.
<code>publisher</code>	string	Your company / lab / individual handle.
<code>license</code>	string	SPDX identifier (MIT, Apache-2.0, Proprietary, ...).
<code>homepage</code>	string	URL to your PDK's docs / repo.
<code>layers</code>	array	Layer table for the Mac app's layer panel. See below.

The layers array

Each entry describes one named GDS layer. The Mac app reads this without importing your Python source, so colors and names are available immediately after install.

```
{
  "layers": [
    {
      "name": "SILC",
      "layer": 9,
      "datatype": 0,
      "color": "#008000",
      "description": "Silicon waveguide core"
    },
    {
      "name": "SILN",
      "layer": 5,
      "datatype": 0,
      "color": "#ff69b4"
    }
  ]
}
```

Field	Type	Required	Description
name	string	yes	Must match a key in your Python LayerTable.
layer	integer	yes	GDS layer number.
datatype	integer	yes	GDS datatype.
color	string	no	Hex color (#RRGGBB). Used in the layer panel.
description	string	no	Tooltip text in the layer panel.

Generate the array from your Python source so it stays in sync:

```
from your_pdk.layers import LAYER

manifest["layers"] = LAYER.to_manifest()
```

Optional fields the app understands

Field	Type	Description
default_view	string	Layer name to default the viewer to when the PDK is the active tech.

Field	Type	Description
default_render_mode	string	"filled" or "outline". Default render mode when this PDK is active.
min_app_version	string	Minimum Lumicron Mac app version. Use sparingly — frustrates users on older builds.

Optional fields ignored by the current app

These are reserved for future use; you can include them but the app v1 won't act on them:

Field	Type	Description
process_node	string	Foundry / process designator ("sky130", "GF45", ...).
wavelength_band	string	"0", "S", "C", "L", "850nm", etc.
simulator_hints	object	Per-simulator overrides (Lumerical, Tidy3D, MEEP).

Example — complete manifest

```
{
  "name": "TinyPhot",
  "module": "tinyphot",
  "version": "0.1.0",
  "lumicron_api_required": ">=0.4",
  "description": "Compact silicon photonics PDK for tutorials.",
  "publisher": "Elyon Semiconductor",
  "license": "MIT",
  "homepage": "https://github.com/elyon-semi/tinyphot",
  "default_view": "SILC",
  "default_render_mode": "filled",
  "layers": [
    {"name": "SILC", "layer": 9, "datatype": 0, "color": "#008000", "description": "Silicon waveguide"},
    {"name": "SILN", "layer": 5, "datatype": 0, "color": "#ff69b4", "description": "Silicon nitride wa"},
    {"name": "OXID", "layer": 1, "datatype": 0, "color": "#87ceeb", "description": "Oxide cladding / e"},
    {"name": "MTL1", "layer": 4, "datatype": 0, "color": "#ffd700", "description": "Metal-1 routing /"}
  ]
}
```

Validation

The Mac app validates manifests before install. Common failures:

Error	Fix
manifest missing required field 'module'	Add the field. Match the directory under source/.
module name 'TinyPhot' invalid (must be snake_case)	Rename to tinyphot (the import name).
lumicron_api_required '>=0.5' not satisfied (have 0.4.0)	Either bump the bundled app or relax your floor.
source directory 'tinyphot' missing	Bundle path mismatch: archive structure is wrong.
layer name 'silicon' has no match in LayerTable	Layer-table key is SILC, not silicon. Fix the manifest entry to match.

You can run the same validator yourself before publishing — `lumicron_api.tools.validate_lumpdk` is the entry point (forthcoming in 0.5).

Glossary

Terms specific to PDK authoring. The end-user glossary in the *Lumicron API User Guide* covers cell, port, pin, route, etc.

Component — A `@pcell`-decorated factory function in your PDK. Returns a `CELL`. The naming is colloquial; technically every PDK component is also a `PCell`.

Continuous stroke — One `p.layer(...)` band in a route profile. Renders as a `FlexRoute` along the entire route, including bends.

Datatype — The second integer in a `(layer, datatype)` pair. Lets multiple semantic layers share one number — drawn (datatype 0), exclusion (1), fill (2), etc.

Default registry — The process-wide `TransitionRegistry` accessed via `lm.default_transitions()`. PDK transition modules register into it on import.

Factory — A zero-argument function that returns a fresh `CELL`. Transitions are stored as factories so the router can re-instantiate per use.

LayerSpec — The minimal layer descriptor: `(layer, datatype)`. `Layer` is a subclass that adds design rules and color.

LayerTable — A named container of `Layer` objects. End users see it as `pdk.LAYER`.

Manifest — `manifest.json` inside a `.lumpdk` archive. Carries name / version / module path / layer map.

Periodic feature — A tooth declared via `p.tooth(cell, period=...)`. The cell is tiled along the route at the given arc-length spacing.

PCell — Parametric Cell. Any function decorated with `@lm.pcell`. Lumicron's `PCells` auto-name distinct parameter combinations into distinct GDS cells.

Process Design Kit (PDK) — The package authored by this guide. Contains layers, profiles, components, and (optionally) transitions for one foundry process or in-house technology.

Process-wide registry — See *default registry*.

Profile — `RouteProfile`. A cross-section recipe — strokes plus periodic features. Built with `@lm.route_profile`.

RouteProfiles — Container collecting route profiles by name. End users see it as `pdk.RP`.

Route profile — See *Profile*.

Sellmeier coefficients — B and C arrays describing material dispersion $n^2(\lambda) = 1 + \sum B_i \cdot \lambda^2 / (\lambda^2 - C_i)$.

Sibling stroke — A continuous stroke other than the primary core. Cladding rectangles, exclusion zones, slot rails — all sibling strokes.

SOI — Silicon-on-insulator. The substrate type for most photonic PDKs.

StackLayer — One physical layer in a `LayerStack`: GDS layer reference, thickness, and optical model.

Subwavelength grating (SWG) — A grating with period much smaller than the operating wavelength, behaving as a homogenized effective index. Built using `p.tooth()` with sub-wavelength period.

Tooth — One periodic feature in a route profile. Declared with `p.tooth(cell, period=...)`.

TransitionRegistry — Maps `(layer_a, layer_b)` → factory function. The router reads it to insert bridge PCells on cross-layer routes.

.lumpdk — The packaging format for PDKs the Lumicron Mac app installs. A zip archive of source code plus `manifest.json`.