



TECHNOLOGY, MADE EXPLICIT.

PDK Authoring Guide

The `lumicron_pdk` Python package

Edition	October 2026
Documentation	6 October 2026
Publisher	Elyon Semiconductor

Lumicron PDK Authoring Guide

© 2026 Elyon Semiconductor. All rights reserved.

Documentation for the Python API bundled with Lumicron.

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

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

Use the documentation matching your installed release.

Table of contents

Welcome	1
What “PDK” means in this guide	1
The PDK developer package	2
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	8
II. Building blocks	9
2. Layers and design rules	10
2.1. The minimum	10
2.2. Design rules	11
2.3. Naming	12
2.4. Datatypes	12
2.5. Programmatic introspection	13
2.6. Stippling and borders	13
2.7. Validation rules	15
2.8. Layer mappings	15
3. The physical stack	16
3.1. A complete stack	16
3.2. Sellmeier coefficients	18
3.3. Layer order	19
3.4. When the stack matters at runtime	19

4. Route profiles	21
4.1. The decorator	21
4.2. A minimal profile - single core	22
4.3. Continuous strokes - <code>p.layer()</code>	24
4.4. Periodic features - <code>p.tooth()</code>	26
4.5. Profile collections	29
4.6. Which knob the route reads	29
4.7. Profile design patterns	30
4.8. Custom bends and propagation models	30
4.9. Shared routing permission	30
5. Components	32
5.1. A simple component	32
5.2. Publishing ports with route profiles attached	34
5.3. What “good ports” look like	36
5.4. Synthesizing geometry	37
5.5. Promoting child ports	37
5.6. Packaged schematic and model metadata	38
5.7. Component design patterns	38
6. Layer transitions	39
6.1. Registering a transition	39
6.2. The factory contract	41
6.3. What a transition cell needs	42
6.4. Where transitions appear in a route	42
6.5. When the registry doesn’t have an entry	42
6.6. What ships with the API	43
6.7. Optional generic elevator	43
III. Packaging & distribution	44
7. Packaging - the <code>.lumpdk</code> format	45
7.1. Anatomy	45
7.2. Declare both PDK names	46
7.3. Build the archive	46
7.4. Loading in Lumicron	47
7.5. Compression	48
7.6. What can go wrong	48
8. Publishing	49
8.1. Versioning	49
8.2. Publishing to PyPI	50
8.3. Publishing a <code>.lumpdk</code>	50
8.4. What to put in your README	51
8.5. CI / release pipeline	51
8.6. Marketing your PDK	52

9. Standard PDK checks and inspection artifacts	53
9.1. Outputs	54
9.2. Extension points	54
9.3. Qualify a technology in a project environment	54
 IV. Cookbook	 56
10. Cookbook - a minimal PDK from scratch	57
10.1. Splitting it into a package	60
10.2. What this minimal PDK is missing	60
10.3. Verifying it works	61
 Appendices	 62
A. Manifest schema	62
A.1. Top-level sections	62
A.2. meta	63
A.3. layers	64
A.4. stack	65
A.5. route_profiles	66
A.6. simulation	66
A.7. components	67
A.8. Compatibility and validation boundary	67
 B. Glossary	 69
 API reference	 71

Welcome

This is the **PDK Authoring Guide** for `lumicron`. 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` 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.pdks.<your_pdk>.all` as `pdsk` followed by component placement and routing - every kwarg they don't pass picks up the right default from the PDK.

Here `<your_pdk>` is `META["package-name"]`. It is a Python identifier and is deliberately separate from the human `META["name"]` shown in Lumicron.

 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.ports["o1"], modulator.ports["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.

The PDK developer package

```
import lumicron as lm
import lumicron_pdk as lpdk
import lumicron_validate as lv
```

Use `lpdk` for layers, process stacks, route profiles, display styles, compact models, packaging and qualification. Use `lm` for PCell geometry and design construction. Use `lv` for fabrication/semantic rule authoring. These packages ship in Lumicron's bundled runtime; this guide does not promise a separate PyPI distribution.

See the [Layout API guide](#) for placement/routing, and the [Validation API guide](#) for DRC/DFX. The standard qualification chapter explains `lpdk.tests.run` and its generated inspection artifacts.

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` 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, Python distribution, local `.lumpdk` release, and the boundary between today's installer and a future 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`'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: the same parameters produce the same cell name and 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 lumicon PDK is a Python package - by convention, `lumicon.pdks.<name>` - exposing these 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 - EdgeCouplerSi, Pad, NxM_MMI, ...
TRANSITIONS	TransitionRegistry	Optional. Maps (layer_a, layer_b) to a bridge PCell.
META	dict	Display name, import package name, version, and publisher metadata.

The conventional file layout:

```
lumicon/pdks/<name>/
__init__.py      # re-exports layers, stack, profiles, and components
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()
meta.py          # META: display/import identity + release metadata
```

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 as lm
import lumicron.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 import segment is the PDK's META["package-name"], not its human display name. Keep both explicit:

```
# meta.py
META = {
    "name": "TinyPhot Research PDK", # shown in Lumicron
    "package-name": "tinyphot",      # used in Python imports
    "version": "0.1.0",
}
```

Display names may contain spaces; package names must be valid Python identifiers. New packages use package-name; the current packer rejects package_name.

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

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

01_define_layers.py

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

Each `lpdk.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 as lm
import lumicron_pdk as lpdk

LAYER = lpdk.LayerTable(
    OXID = lpdk.Layer(1, 0, color="#87ceeb"),
    SILC = lpdk.Layer(9, 0, color="#008000",
                      min_width=0.2, min_spacing=0.2, min_radius=5.0),
    SILN = lpdk.Layer(5, 0, color="#ff69b4",
                      min_width=0.4, min_spacing=0.3, min_radius=10.0),
    MTL1 = lpdk.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_gds("layer_swatch.gds")
```

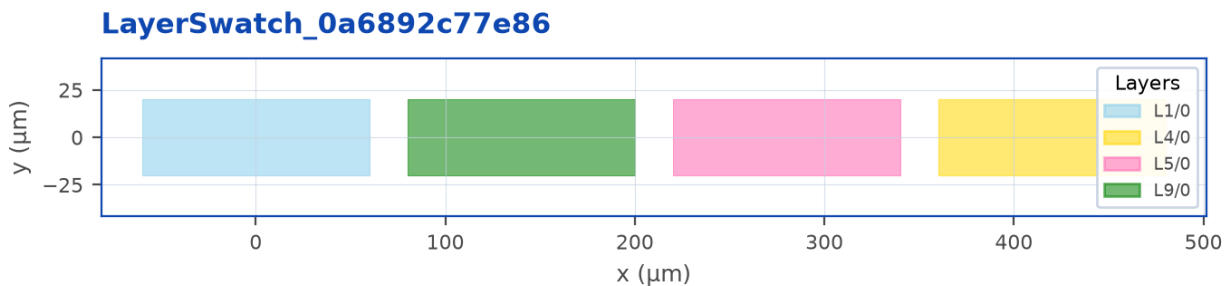


Figure 2.1.: Output of `01_define_layers.py` - one swatch per layer.

Two things this example introduces:

1. `lpedk.Layer(layer, datatype, ...)` - a `LayerSpec` with attached design rules. Use it anywhere the API takes a `LayerSpec`.
2. `lpedk.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 can carry process guidance alongside its physical layer number:

Field	Where it's used
<code>min_width</code>	Minimum-width metadata; declare a named deck rule below for native DRC.
<code>min_spacing</code>	Minimum-spacing metadata; declare a named deck rule below for native DRC.
<code>min_radius</code>	Required minimum physical curvature radius; invalid rendered geometry can fail finalization.

Add them as you have them - partial coverage is fine. These fields do not automatically create a native DRC deck.

```
SILC = lpdk.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 = lpdk.LayerTable(
    SILC      = lpdk.Layer(9, 0, ...),
    SILC_EXCL = lpdk.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).

2.6 Stippling and borders

Choose the display beside the physical layer, inside `LayerTable`:

```
import lumicron as lm
import lumicron_pdk as lpdk

LAYER = lpdk.LayerTable(
    CORE=lpdk.Layer(1, 0, color="#8183FF",
        stipple=lpdk.Stipple.DIAGONAL_UP, opacity=0.35,
        border=lpdk.Border(color="#C4C5FF", width_px=2,
            style="dashed", opacity=1)),
    CLAD=lpdk.Layer(2, 0, color="#58B89A", stipple=lpdk.Stipple.SPARSE_DOTS),
)
```

Fill opacity and border opacity are independent numbers from zero to one. `border=None` retains the ordinary solid outline. Use `lpdk.Border(style="none")` to hide it, or `lpdk.Stipple.EMPTY` for an outline without fill. Border color defaults to the layer color. Existing declarations without these options keep their defaults.

The collection below is available as `lpdk.Stipple` constants. Each mask bit occupies four drawable pixels in the native viewer; the catalogue enlarges the bits for reading.

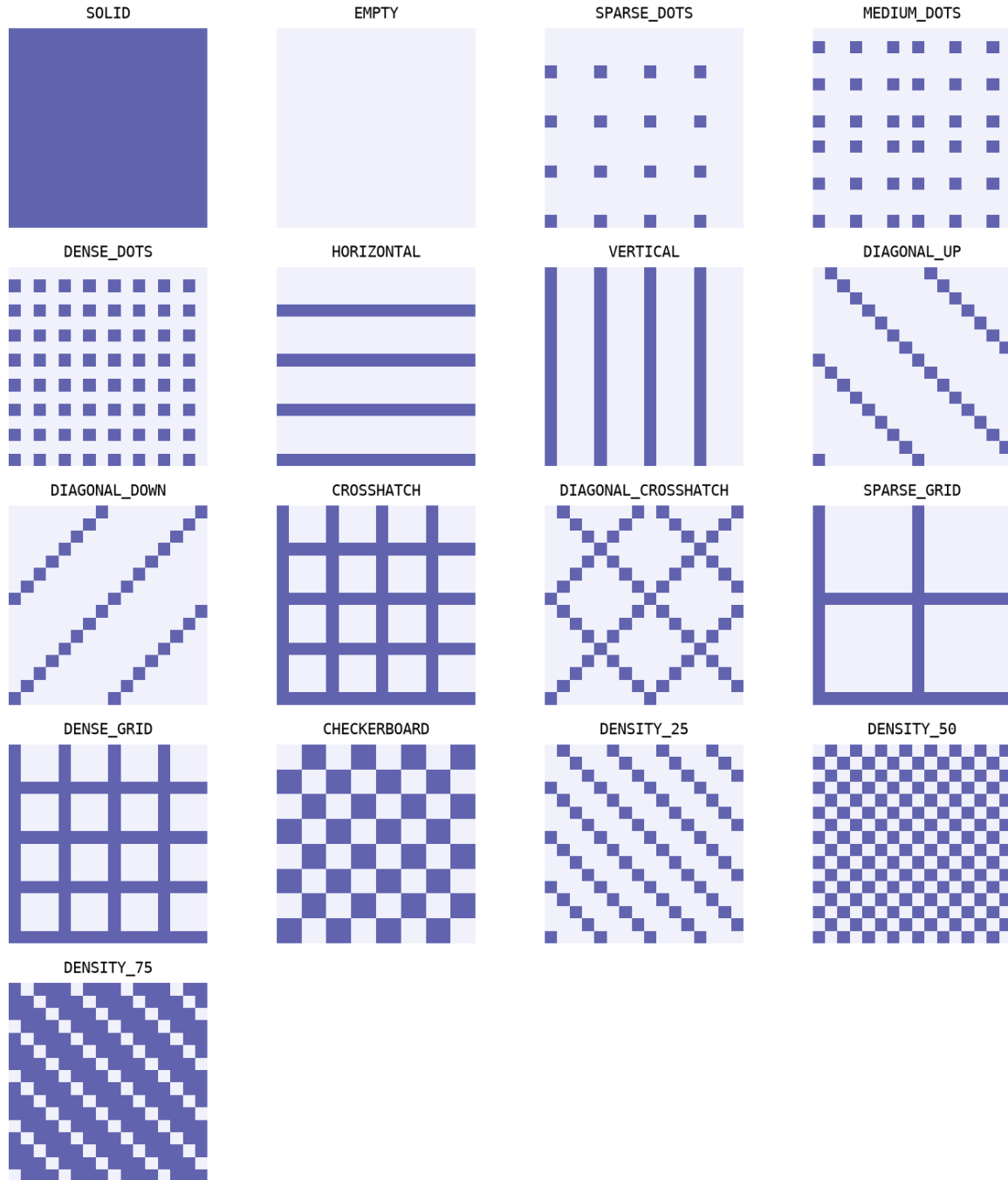


Figure 2.2.: Named stipple collection.

`lpgk.Border` accepts `solid`, `dashed`, `dotted`, `dash-dot`, or `none`. `width_px` is a positive drawable-pixel width up to 64. Named dash lengths are (6, 3), (1, 3), and (6, 3, 1, 3) respectively. These are display dimensions, not micrometers: they stay the same size while zooming. On a Retina display, two drawable pixels normally occupy one screen point. Dash distance follows each polygon contour, restarts for each contour, and does not depend on camera translation.

Corners use a miter limited to four times the half-width; dash ends are flat.

Custom patterns and borders can be shared by several layers:

```
MY_STIPPLE = lpdk.Stipple(rows=(
    "*.....", ".*.....", "..*.....", "...*.....",
    "....*...", ".....*", ".....*", ".....*",
))
MY_BORDER = lpdk.Border(width_px=2, dashes_px=(8, 3, 2, 3))
```

Masks require eight rows of eight * or # (on) and . (off) characters. Custom dash arrays require an even count of 2–16 positive, finite on/off lengths, each at most 4096 drawable pixels. Choose a named style or dashes_px, not both. Invalid declarations raise errors before packaging.

The Python declarations own the style. `LayerTable.to_manifest()` projects it into `manifest.json`; no separate `display.json` is needed. Existing JSON-only PDKs still work. If both sources declare the same layer, their complete display objects must be identical; conflicting declarations fail packaging. New stipple, opacity and border fields are excluded from geometry serialization and authored identity input. They travel with the PDK, not raw GDS or a standalone layout.

2.7 Validation rules

Author named DRC rules with `lumicron_validate.drc`. Export DRC from the PDK root so packaging includes its schema-v1 deck. See [conventional rule authoring in the Validation API guide](#).

2.8 Layer mappings

`lpdk.LayerMap` owns name-to-layer mappings used by PDK and serialization workflows. Ordinary designers can use `lm.LayerSpec(layer=1, datatype=0)` for raw geometry or consume named layers from their environment. PDK authors define richer named layers through `lpdk.LayerTable`; validation-rule authoring remains in `lumicron_validate`.

CHAPTER 3

The physical stack

The LayerStack records authored layer thicknesses, vertical positions, and optical material data. Lumicron packages these declarations for technology inspection and simulation consumers. A stack declaration alone does not run a field solver; Maple execution remains disabled in this alpha.

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

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 as lm
import lumicron_pdk as lpdk

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

```

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

STACK = lpdk.LayerStack(
    lpdk.StackLayer("BOX", layer=LAYER.BOX, thickness=2.0, sellmeier=SiO2),
    lpdk.StackLayer("core_si", layer=LAYER.SILC, thickness=0.22, sellmeier=Si),
    lpdk.StackLayer("core_sin", layer=LAYER.SILN, thickness=0.4, sellmeier=SiN),
    lpdk.StackLayer("clad", layer=LAYER.CLAD, thickness=2.0, sellmeier=SiO2),
    lpdk.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__":
    StackPreview().to_gds("stack_preview.gds")

```

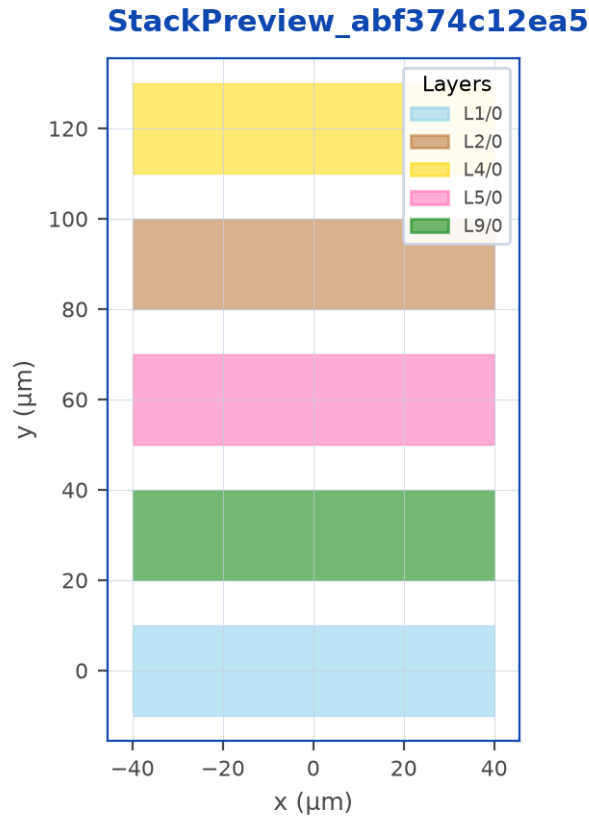


Figure 3.1.: Output of `02_layer_stack.py` - process layers as horizontal swatches.

The material and stack building blocks:

Type	Purpose
Sellmeier	Coefficients for $n^2(\lambda) = 1 + \sum B_i \cdot \lambda^2 / (\lambda^2 - C_i)$.
TabulatedIndex	Validated wavelength samples; use bulk material data for a stack layer.
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.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 a declared constant real index approximation, pass `n=` directly:

```
lpdk.StackLayer("cladding", layer=LAYER.OXID, thickness=1.0, n=1.44)
```

Use only one of `n`, `sellmeier`, or `tabulated` on each layer. `tabulated` must describe a bulk material, not a waveguide effective index. If optical properties are unavailable, leave them unspecified; `n=0` is not a special declaration of a metal or perfect conductor.

3.3 Layer order

When `z_start` is omitted, the first layer starts at zero and each following layer starts at the preceding layer's end. An explicit `z_start` is preserved, including zero or negative positions, and can express a gap or overlap. `z_min` and `z_max` report the resulting stack envelope. A typical authored SOI sequence might be:

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

Routing geometry is declared by profiles and components; the stack provides separate physical technology metadata. It is serialized into the PDK manifest and available through `pdsk.STACK`. Use `pdsk.STACK.print_stack()`, iterate its layers, or inspect a named layer's thickness, `z_start`, and `n_at(wavelength)` value. A `LayerStack.cross_section(at=...)` method is not part of the current API.

Publish only material and dimensional values supported by your process data. Missing optical data should remain missing, rather than being filled with a placeholder that could be mistaken for a validated simulation input.

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.

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 lumicon as lm
import lumicon_pdk as lpdk

@lpdk.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)
```

Decorator settings:

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	Required minimum rendered curvature radius, distinct from the layout footprint. Defaults to radius.
port_layer	Optional fallback for PORT.layer. 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

03_route_profile_basic.py

```

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

The `@lpdk.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 as lm
import lumicron_pdk as lpdk

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

@lpdk.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 = lpdk.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.ports["in"], c.ports["out"])
    return c

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

```

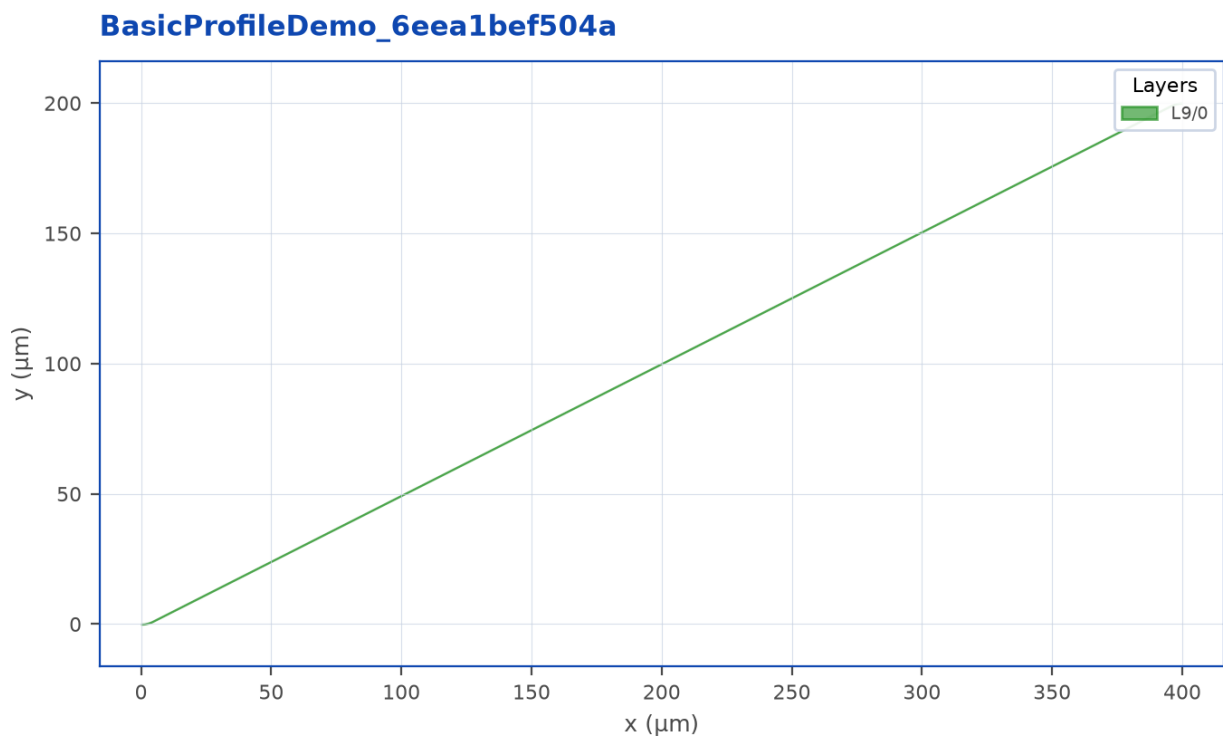


Figure 4.1.: Output of 03_route_profile_basic.py.

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:

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 lumicon as lm
import lumicon_pdk as lpdk

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

@lpdk.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.ports["in"], c.ports["out"])
    return c

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

```

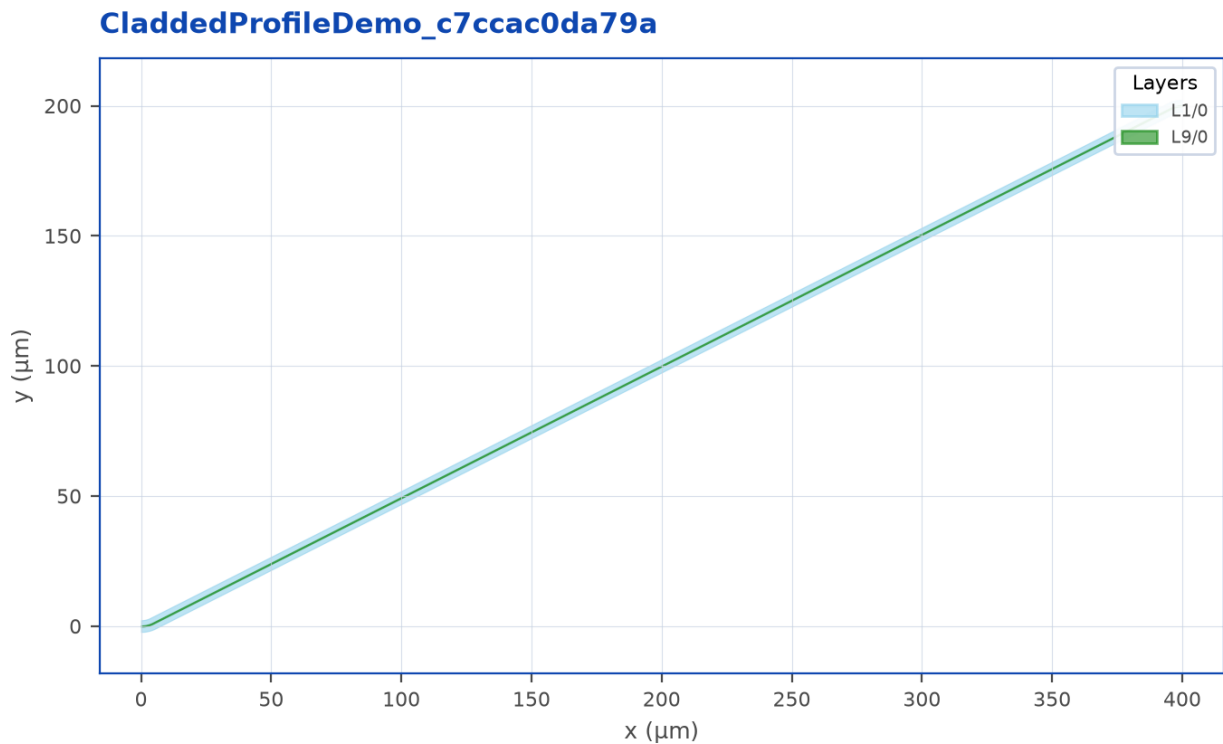


Figure 4.2.: Output of 04_route_profile_cladded.py - silicon core with oxide cladding.

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

4.3.1 Scalar PORTs and multistroke profiles

PORT.width is one scalar width in micrometres. It is not a list of rail apertures, a slot gap, or the full envelope of a multistroke cross-section. PORT(route_profile=profile) defaults to profile.

port_width and profile.port_layer; explicit width= and layer= override those defaults. The connection-layer declaration is named port_layer, not connection_layer.

For routing, a centered stroke on port_layer is selected as primary, even when another layer was declared first. Its width follows the incoming scalar PORT width. If no such centered stroke exists, the first stroke remains primary and keeps its declared offset and width. Other strokes retain their own widths and offsets. Waveguide's generated PORTs use that primary stroke's width and layer; an explicitly constructed PORT still uses its own defaults.

Generic endpoint tapers compare each endpoint's scalar width with the primary route width. They are centered linear polygons on the primary layer; they do not synthesize a transition for two separated rails. For example, declaring a 0.2 μm gap as port_width while the primary rail is 0.25 μm causes generic tapers under the default policy. RoutingPolicy(auto_taper=False) rejects that mismatch instead of changing the rails.

A bounded constant-rail pattern separates the gap from the scalar width:

```
@lpgk.route_profile(port_layer=LAYER.SILC, port_width=0.25,
                    routing_policy=lpgk.RoutingPolicy(auto_taper=False))
def fixed_slot(p):
    rail, gap = 0.25, 0.2
    p.layer(LAYER.SILC, width=rail, offset=(gap + rail) / 2)
    p.layer(LAYER.SILC, width=rail, offset=-(gap + rail) / 2)
```

Waveguide(points=..., profile=fixed_slot) emits the two rails. A same-profile straight Route between matching 0.25 μm scalar PORTs also emits those rails without generic tapers. This demonstrates geometry; it does not establish a general multirail interface, slot mode, arbitrary slot taper, or LVS support for a PORT centered in the gap. Keep reference examples within that boundary. Two different profiles on the same layer require an explicit PDK transition; the layer-pair registry does not infer a strip-to-slot transition.

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).

Arg	Purpose
phase	Arc-length of the first tooth. Default period/2 - center of slot 1.
span_bends	If False, skip teeth that would land inside a bend arc.

Waveguide and CELL.Route use the same rendered-spine placement mechanism. Distance is measured on the unoffset reference centerline, including circular or Euler bends; a tooth's positive offset is to its left. The tooth cell's local origin is placed on that offset line and its local +x follows the sampled tangent. Phase continues across resolved sections of a Route, so a shared section boundary does not emit two teeth. Negative phase advances by whole periods to the first nonnegative station. A station exactly at the final endpoint is included. Whole tooth cells are retained: geometry extending beyond the endpoint is not clipped.

span_bends=False skips stations on curved sampled segments without restarting phase after the bend. Circular/Euler placement uses the rendered polyline and its curvature mask at the default 0.001 μm sampling tolerance. Auxiliary teeth do not add to the reported route centerline length. GDS retains their hierarchy and semantic route ownership; OASIS retains geometry, without a Lumicron semantic sidecar.

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 μm for driven waveguides.
- **Mode-converter periodic claddings** - sub-wavelength teeth for SWG cross-sections.

A working grating profile:

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 as lm
import lumicron_pdk as lpdk

LAYER = lpdk.LayerTable(
    SILC = lpdk.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

@ltpdk.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.ports["in"], c.ports["out"])
    return c

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

```

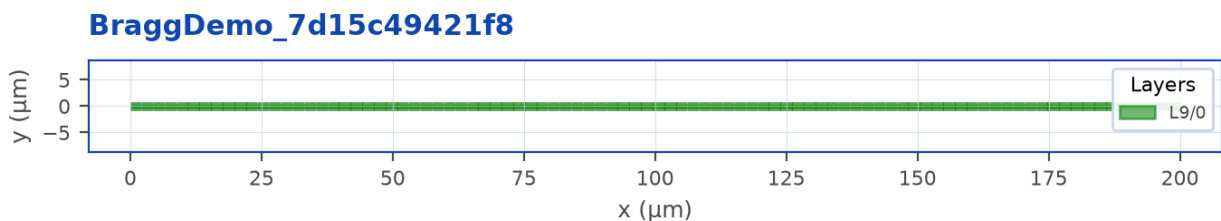


Figure 4.3.: Output of 05_route_profile_grating.py.

⚠ 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 ($\lambda/2n \approx 320$ nm) gratings, prefer `span_bends=False` and

design your route to keep grating regions on straights. For coarse-period features (period > 5 × bend radius) the default works.

Warning

Custom bend PCells. A used custom bend PCell combined with periodic features is rejected by the existing profile compatibility check. It does not silently omit teeth. An unused custom-bend default on a straight path does not prevent periodic emission. Use supported circular or Euler bends for curved periodic profiles.

4.5 Profile collections

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

```
RP = lpdk.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(profile=...)` 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_oxide_clad` is not - the user shouldn't have to know about the cladding to pick a profile.

4.8 Custom bends and propagation models

The decorator also accepts `bend=` and `propagation=`. A custom bend can be a PCell factory or supplied CELL; its geometry and ports establish its footprint, not a presumed circular or Euler shape. Report a custom cell's identity; do not assign it an analytic radius or path length unless the PDK provides corresponding evidence. `radius_min` is a required physical curvature constraint for geometry where curvature is available, not only a warning.

A declared propagation model supplies circuit behavior for a route of known measured length. Use a declared `CompactModel` or effective-index `TabulatedIndex`; material core index alone does not establish a solved effective mode index. Unknown custom-bend length must remain unknown rather than silently using a circular approximation.

4.9 Shared routing permission

The PDK owns permission to synthesize generic width transitions. Share an immutable policy across all profiles of the technology:

```
PDK_ROUTING = lpdk.RoutingPolicy(auto_taper=False)

@lpdk.route_profile(port_layer=LAYER.SILC, port_width=0.5,
                    radius=10, routing_policy=PDK_ROUTING)
def strip(p):
    p.layer(LAYER.SILC, width=p.port_width)
```

True (the default) permits the existing linear taper mechanism. False blocks generic taper generation anywhere on a route using a restrictive endpoint, global or section profile. It does not globally mutate other PDKs: every relevant PDK profile must carry the shared policy. The policy is projected into PDK inspection/package metadata and retained in typed Python snapshots. Legacy profiles continue to permit tapers. Raw layer-only ports carry no PDK permission evidence.

The existing custom cell= taper options accept a straight two-port PDK CELL, including off-origin geometry. Its declared widths, span, layers, port type and facing directions must match the requested transition. Explicit device geometry is allowed under a restrictive generic-synthesis policy. There is no new taper-factory selection protocol or universal transition physics. Same-layer profile-to-profile transition discovery is not established; author those devices explicitly. The existing layer-pair registry is described in the transitions chapter.

A policy is design permission, not simulation or process qualification. `port_width` remains a default value, not a rule to widen every narrower port.

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 records definition identity, parameters, and cell naming.

5.1 A simple component

`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 lumicon as lm
import lumicon_pdk as lpdk

LAYER = lpdk.LayerTable(
    MTL1 = lpdk.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_gds("pad_demo.gds")

```

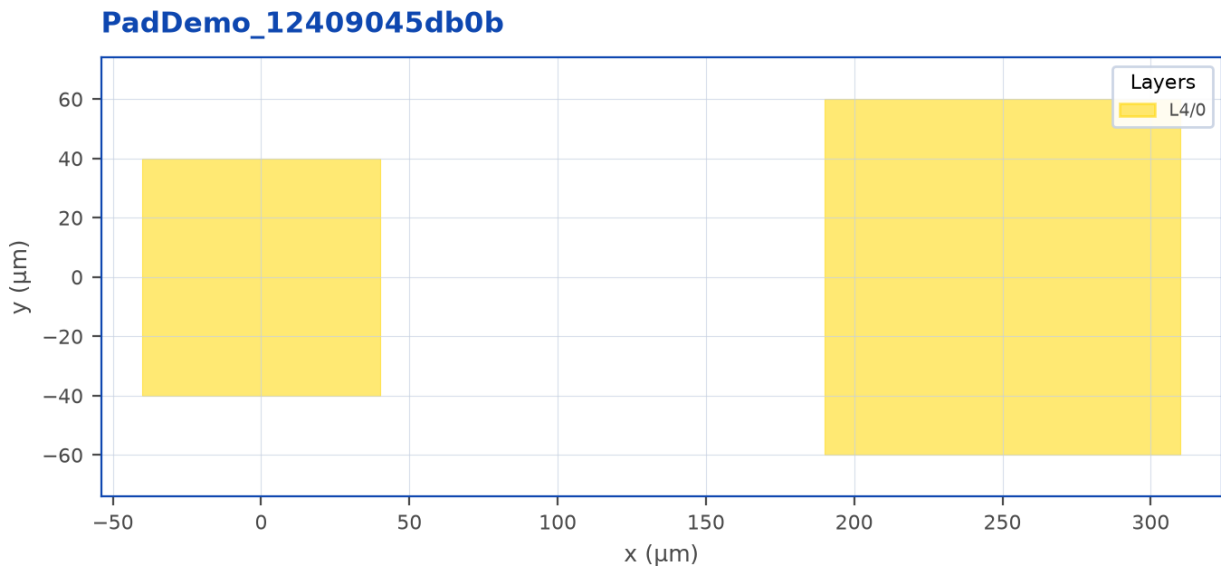


Figure 5.1.: Output of 06_simple_component.py - two pads of different sizes.

What `@pcell` is doing for you:

- **Stable identity.** The name is derived from the qualified factory, source, PDK identity, and effective typed parameters, including defaults. Imported GDS content also contributes to

identity.

- **Fresh construction.** Each call executes the factory and returns its constructed cell. Repeated calls are not a cache of the same mutable CELL object. To reuse one constructed definition, keep that cell and add it to the parent more than once.

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.

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.ports["o1"], b.ports["i1"])`` with no kwargs and
the route automatically stamps the right layers, claddings, and
periodic features.
"""

import lumicron as lm
import lumicron_pdk as lpdk

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

@lpdk.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 μm tip to a 0.5 μm 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 → 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

@lm.pcell
def CouplerDemo():
    c = lm.CELL("CouplerDemo")
    cpl = c.add(EdgeCouplerSi(length=100))
    c.Place(cpl).at((0, 0))

    # Publish a destination port on the chip and route from coupler.o1.
    # No `radius=` / `profile=` – both come from the port's profile.
    c.add(lm.PORT("out", position=(500, 100), direction=180,
                route_profile=silc_strip))
    c.Route(cpl.ports["o1"], c.ports["out"])
    return c

if __name__ == "__main__":
    CouplerDemo().to_gds("coupler_demo.gds")

```

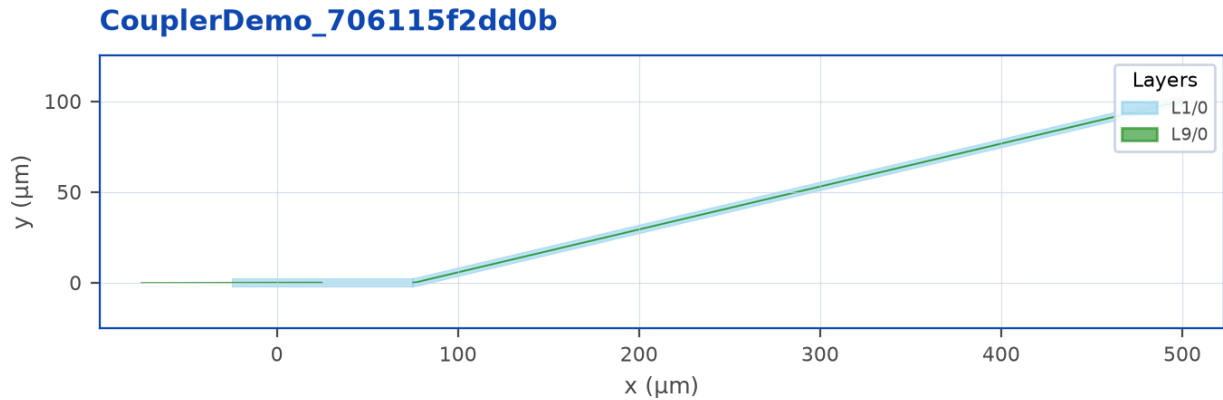


Figure 5.2.: Output of 07_component_with_profile.py - coupler plus auto-routed exit.

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.ports["o1"], chip.ports["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` 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 Packaged schematic and model metadata

The existing packager preserves declared symbol hints and named compact-model metadata in legacy/application integrations. Review any `port_resolution_error` or `model_resolution_error` before publishing. Geometry alone does not supply a circuit model. Experimental `CELL.Model/CELL.Simulate` syntax is archived; this guide does not teach it as a future public simulation API. Existing packaged models remain readable through deprecated compatibility forwarding.

5.7 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. Use a readable `lm.CELL("ComponentName")` inside the factory; `@pcell` derives the final bounded name from the factory name and definition identity. 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).ports["p"], MZI(length=200).ports["i1"]`). Then implement the components to make the tests pass. The result is a PDK whose API was designed from the user's seat.

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

`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 as lm
import lumicron_pdk as lpdk

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

@lpdk.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)
```

```

@ltdk.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 ltdk.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 = ltdk.TransitionRegistry()
TRANSITIONS.register(LAYER.SILC, LAYER.SILN, silc_siln_elevator)
ltdk.default_transitions().register(LAYER.SILC, LAYER.SILN, silc_siln_elevator)

@ltdk.pcell
def TransitionDemo():
    """A route whose endpoints are on different layers – the registered
    elevator drops in automatically."""
    c = ltdk.CELL("TransitionDemo")
    c.add(ltdk.PORT("in", position=(0, 0), direction=0,
                    route_profile=silc_strip))
    c.add(ltdk.PORT("out", position=(600, 0), direction=180,
                    route_profile=siln_strip))
    c.Route(c.ports["in"], c.ports["out"], transition_offset=300)
    return c

if __name__ == "__main__":
    TransitionDemo().to_gds("transition_demo.gds")

```

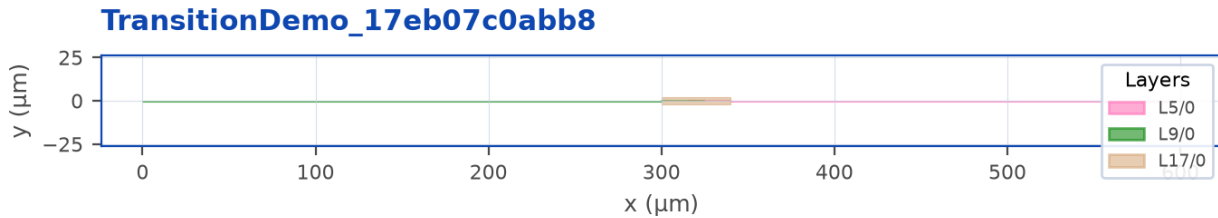


Figure 6.1.: Output of 08_transition_pcell.py - auto-inserted SiLC-to-SiLN elevator.

Two registries appear:

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

lpdk.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 lpdk.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 optional `lpdk.PhotonicElevator` provides two inverse tapers with an overlap region; use it only when that con-

struction fits the process. A custom CELL with the required ports can implement different transition physics.

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 places the transition's entry N μm from port_a, along that port's outgoing direction. The default is 50.0 μm ; it is not a midpoint calculation.

For section-level layer changes (.style(profile=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-to-electrical transitions, or photonic transitions across very different stacks (silicon-to-III-V), aren't well-defined - 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(SILC, SILN, factory, symmetric=True)` registers both directions; `True` is the default. The reverse entry uses the same factory with reversed placement. Pass `symmetric=False` when only the declared direction is supported. Registration authorizes geometry placement; it does not establish reciprocal optical performance.

6.6 What ships with the API

Two transition primitives are bundled:

- `lpdk.PhotonicElevator`: optional generic inverse-taper geometry for a suitable PDK or reference test.
- `lpdk.TransitionRegistry`: 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.

6.7 Optional generic elevator

`lpdk.PhotonicElevator` is the existing generic inverse-taper-style convenience PCell. It can provide reference/test geometry or a transition when its construction matches the platform. It does not prescribe every PDK's transition physics. A PDK may provide a completely custom transition PCell. Lumicron owns transition reasoning; the PDK owns transition physics.

The generic elevator contains polygons, without a certified optical centerline. Its geometry can be exported in an automatic transition route, but the total optical route length remains unavailable. Neither port separation nor the transition's bounding box supplies the missing path-length evidence.

Part III.

Packaging & distribution

CHAPTER 7

Packaging - the .lumpdk format

A .lumpdk is a zip archive containing a generated manifest.json, the Python source inside one PDK package, declared model resources, package-local GDS assets, and an optional drc.json. Load the archive into a Lumicron project using File > Load PDK or Change PDK.

You do not need a .lumpdk to publish a PDK to PyPI. Use a wheel for ordinary Python installation and a .lumpdk for one-click installation in the Mac app; both can be produced from the same source package.

7.1 Anatomy

lumicron_pdk.packaging writes the **contents of the imported package** at the archive root:

```
tinypht.lumpdk
├─ manifest.json      # generated by pack_pdk
├─ __init__.py        # package root after app extraction
├─ all.py
├─ meta.py            # META: display/import identity + version
├─ layers.py
├─ profiles.py
├─ components.py
├─ transitions.py
├─ pcells/            # optional Python subpackages are retained
│   └─ ring.py
├─ models/            # declared compact-model / optical data files
│   └─ mmi.csv
├─ layouts/           # package-local .gds / .gdsii resources
│   └─ bend.gds
└─ drc.json           # optional; generated from DRC or copied from JSON
```

Do not wrap these files in `source/tinyphot/` inside the archive. The app already creates the destination directory named by `package-name` in its private cache. Use package-relative imports and resource paths so the package remains portable after extraction.

The packer includes Python source, declared compact-model data, source CSVs used by declared TabulatedIndex models, all package-local `.gds` / `.gdsii` assets, and optional root `drc.json`. Arbitrary documentation and unrelated data files are not copied automatically. Referenced model resources must stay inside the package. Optical CSV and GDS resources are checked against their recorded content hashes; rebuild after changing them.

The archive filename must be exactly `<package-name>.lumpdk`; store release versions in `META["version"]` and, if needed, versioned release directories. A filename such as `tinyphot-0.1.0.lumpdk` is rejected when `package-name` is `tinyphot`.

7.2 Declare both PDK names

Put a META dictionary in `meta.py`:

```
META = {
    "name": "TinyPhot Research PDK",
    "package-name": "tinyphot",
    "version": "0.1.0",
    "vendor": "Example Lab",
    "description": "Compact silicon photonics PDK for tutorials.",
    "units": "um",
}
```

name Human-facing identity shown in project PDK panels. Spaces and capitalization are fine.

package-name Python identifier used for the extracted folder and `import lumicron.pdks.tinyphot.all as pdk`. It must satisfy Python's identifier rules. `package_name` is rejected; use the canonical `package-name` key.

version PDK release version displayed by Lumicron and used by your release process.

Declare both identities explicitly. If `package-name` is absent, the packer uses the source package directory name. An invalid identifier, a Python keyword, or the legacy `package_name` key causes packing to fail; no display-name sanitization is performed by the packer.

7.3 Build the archive

The package must be importable in the Python environment that runs the packer:

```
python -m lumicron_pdk.packaging tinyphot --output-directory dist
```

Or from Python:

```
from lumicron_pdk.packaging import pack_pdk

pack_pdk("tinyphot", output_directory="dist")
```

For a `src/-layout` repository, install the package editable or add `src/` to `sys.path` in a small build script before calling `pack_pdk`. Pass the package itself, not its `.all` re-export module.

The packer imports these public namespaces when available:

- `META` from `<package>.meta`;
- `LAYER` and `STACK` from the package root;
- `RP` from the package root;
- decorated factories from `<package>.components`, or the root when that module is absent;
- ports and named circuit-model declarations from constructed component cells;
- declared data resources and package-local GDS files;
- optional DRC deck exported at the package root, or `drc.json` beside `__init__.py`;
- layer display declarations in `LAYER`, or legacy `display.json`.

It serializes them into top-level manifest sections: `meta`, `layers`, `stack`, `route_profiles`, `simulation`, and `components`, with `component_schema_version: 1`. Appendix A documents that generated schema. The app reads this packaged component contract; it does not reconstruct the schematic library by guessing from Python source.

Warning

The packer imports the PDK. Keep package import deterministic and free of network access, GUI startup, or file-writing side effects. It also evaluates component defaults and bounded boolean parameter variants to collect ports and models. Factories must construct cells without opening windows, writing exports, or launching simulations. Registration imports such as transitions are fine.

7.4 Loading in Lumicron

Load the package through **File > Load PDK** or **Change PDK** in the project inspector. Verify its displayed identity, component catalog, layer map, and explicit Python import. Test upgrades against a copy of a representative project and retain the previous archive. The archive's metadata does not by itself prove that its Python components are executable.

7.5 Compression

Use ZIP stored entries or ordinary Deflate. Python's `zipfile.ZIP_DEFLATED` is the default used by `pack_pdk`. The app's bounded reader does not support BZIP2 or LZMA entries.

7.6 What can go wrong

Symptom	Likely cause / fix
PDK appears but scripts cannot import it	Declare a valid <code>META["package-name"]</code> , use relative internal imports, and test the extracted package.
Import uses the display name with spaces Installed source or model data is missing	Import by <code>package-name</code> , never by name. Keep required code and declared model resources inside the imported package; unrelated files are not copied automatically.
An old PDK version is still active	Load the intended archive into the project and verify its version; runtime extraction is cached by archive content.
Layers have generic colors	Export <code>LAYER</code> from the package root and use <code>LayerTable</code> entries with colors.
DRC tool reports no installed deck	Export a <code>drc.Deck</code> as DRC from the package root, retain <code>drc.json</code> , or pass an explicit deck.
A component is unavailable in the schematic library	Inspect its packaged parameters, ports, and model-resolution errors; displaying layer colors does not validate component execution.

Before release, install the `.lumpdk` in Lumicron and run a clean script containing the canonical long import. A successful import plus one generated GDS is the minimum end-to-end packaging gate.

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.

Keep distribution identity separate from product identity: PyPI's distribution name can be `lumicron-pdk-tinyphot`, the Python/`.lumpdk` package name can be `tinyphot`, and the human display name can be "TinyPhot Research PDK". Record the latter two in `META["package-name"]` and `META["name"]` respectively.

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; use an upper bound such as `>=1.2,<2` if accepting compatible updates. A lower bound alone also permits future breaking releases.

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"
# Pin the API distribution version validated by your release tests.
dependencies = ["lumicron"]

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

Two non-obvious bits:

- **Do not replace API package files.** The API owns `lumicron/__init__.py` and `lumicron/pdks/__init__.py`; do not ship replacements in your PDK wheel. Package discovery with `namespaces = true` can include your nested PDK directory without those parent files. Test the wheel in a clean environment alongside the matching API distribution.
- **Source layout.** Put your code at `src/lumicron/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

Build from the importable package itself:

```
python -m lumicron_pdk.packaging lumicron.pdks.tinyphot \
--output-directory dist/1.2.0
```

This writes `dist/1.2.0/tinyphot.lumpdk`; the filename must match `package-name`. Distribute the archive with its release notes and tested Lumicron/API build identity. Users load it from the app's **Load PDK** command. A marketplace is not part of this alpha.

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** - `python -m lumicron_pdk.packaging <package> --output-directory dist`.
5. **Publish to PyPI** - `twine upload dist/*.whl`.
6. **Attach the .lumpdk to the GitHub release.**

Measure build and validation time for your 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 and test a `lumicron` version. Put the dependency range in `pyproject.toml` for wheels and state the tested Lumicron app/API versions in release notes. The current `.lumpdk` installer does not enforce a `lumicron_required` manifest field, so CI and explicit release notes are the compatibility gate today.

CHAPTER 9

Standard PDK checks and inspection artifacts

Use the test runner from the same bundled runtime that will consume the PDK:

```
from lumicron_pdk import tests

report = tests.run("my_pdk", output_directory="/private/tmp/my-pdk-qualification")
print(report.summary())
print(report.counts)
```

The package must be importable. Its source directory and output must be disjoint. The runner executes trusted PDK Python in a fresh subprocess so an already imported module cannot masquerade as changed source. It is not a sandbox for untrusted PDKs.

`tests.available()` lists standard groups. `checks=("structure", "components")` selects groups. The default runs all of them:

Group	Evidence
structure	Existing manifest extraction, declared layers and component-resolution errors
packaging	Existing packer and a <code>.lumpdk</code> artifact
components	Each defaultable PCell exported to GDS, schema-2 sidecar and <code>.lum</code> ; finite port properties
layer-display	One tile per declared layer; install the generated PDK to inspect styles
route-profiles	One 100 μm straight route per declared profile
qualification	Simulation and fabrication checks explicitly skipped without evidence and acceptance criteria

Required component parameters without standard defaults are skipped. A component exception is a failed result for that component; it does not hide the other checks. Missing layer declarations fail the structural check. No declared components/profiles produces skipped checks.

9.1 Outputs

The output contains `results.json`, `results.txt`, `worker.log`, the resolved `manifest.json`, `package/`, `components/`, `layers/` and `profiles/` when applicable. Component/profile filenames have readable names plus bounded suffixes. Each result identifies its group, subject, status, reason, warnings and artifact paths.

`report.is_current(package_directory)` checks the PDK source/asset snapshot and API implementation digest. `report.artifacts_intact()` verifies generated artifact digests. Source mutation during execution invalidates the report. Version-control directories and Python bytecode caches are excluded; external dependencies and process/fabrication history are not PDK input evidence. Keep authored resources in the package.

Rerunning replaces only a prior runner-owned output folder. Unrelated output is refused. Staging and rollback protect ordinary replacement failures; power-loss atomicity across directory replacement is not claimed. A failed rerun retires the preceding inspection artifacts.

These are structural checks and inspection layouts. They do not establish optical accuracy, bend loss, manufacturing suitability or process qualification. Inspect the generated layouts in Lumicron and provide separate simulation/measured evidence for those questions.

9.2 Extension points

Standard checks live in independent functions in the runner's private check registry. Future standard groups can be added without changing the orchestration or result format. This phase does not expose a general plugin or sweep language. For fabrication rule authoring and bounded semantic checks, see the [Validation API guide](#).

9.3 Qualify a technology in a project environment

A general engineering package can consist of `lumicron.toml` and `technology.py`. See the Layout guide's *Packages and project environments* chapter and the permanent Examples/Packages/SimplePDK example. It declares technology without components, a mandatory stack, or an archive descriptor. Component libraries can depend on it without copying the layer table.

Qualify an exact resolved technology using the same runner:

```
from lumicron.packages import resolve
from lumicron_pdk import tests

environment = resolve("/path/to/DemonstrationTapeout")
report = tests.run("example.simple", environment=environment,
                  output_directory="/path/outside/packages/qualification")
print(report.summary())
```

The report includes package/version/process/environment binding. Currentness requires `report.is_current(package_directory, environment=environment)`; changed dependencies invalidate that evidence too. Qualification does not edit package source. Legacy importable PDK qualification is unchanged.

Generated GDS, semantic sidecars and `.lum` inspection artifacts are supported. Legacy `.lumpdk` packaging of a general engineering-package dependency graph is reported **skipped**, not passed. Distribution of these source packages uses their manifest and locked dependencies. Simulation and fabrication qualification remain unavailable unless the corresponding check has real evidence.

Part IV.

Cookbook

CHAPTER 10

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.

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

import lumicron as lm
import lumicron_pdk as lpdk

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

# — Route profiles —————
@lpdk.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)
```

```

@lpgk.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 = lpgk.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))
    c.add(lm.PORT("o1", position=(0, 0), direction=180,
                  width=0.5, route_profile=silc_strip))
    c.add(lm.PORT("o2", position=(length, 0), direction=0,
                  width=0.5, route_profile=silc_strip))
    return c

# — Transitions —————
def silc_siln_elevator():
    return lpgk.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.OXID, cladding_margin=2.0,
        route_profile_a=silc_strip,
        route_profile_b=siln_strip,
    )

TRANSITIONS = lpgk.TransitionRegistry()
TRANSITIONS.register(LAYER.SILC, LAYER.SILN, silc_siln_elevator)

```

```

lpgk.default_transitions().register(LAYER.SILC, LAYER.SILN, silc_siln_elevator)

# — Demo cell using the PDK we just defined —————
@lm.pcell
def TinyPhotDemo():
    c = lm.CELL("TinyPhotDemo")
    wg1 = c.add(Waveguide(length=80))
    wg2 = c.add(Waveguide(length=80))
    c.Place(wg1).at((0, 0))
    c.Place(wg2).at((300, 200)).rotate_by(180)
    c.Route(wg1.ports["o2"], wg2.ports["o2"])
    return c

if __name__ == "__main__":
    TinyPhotDemo().to_gds("tinyphot_demo.gds")

```

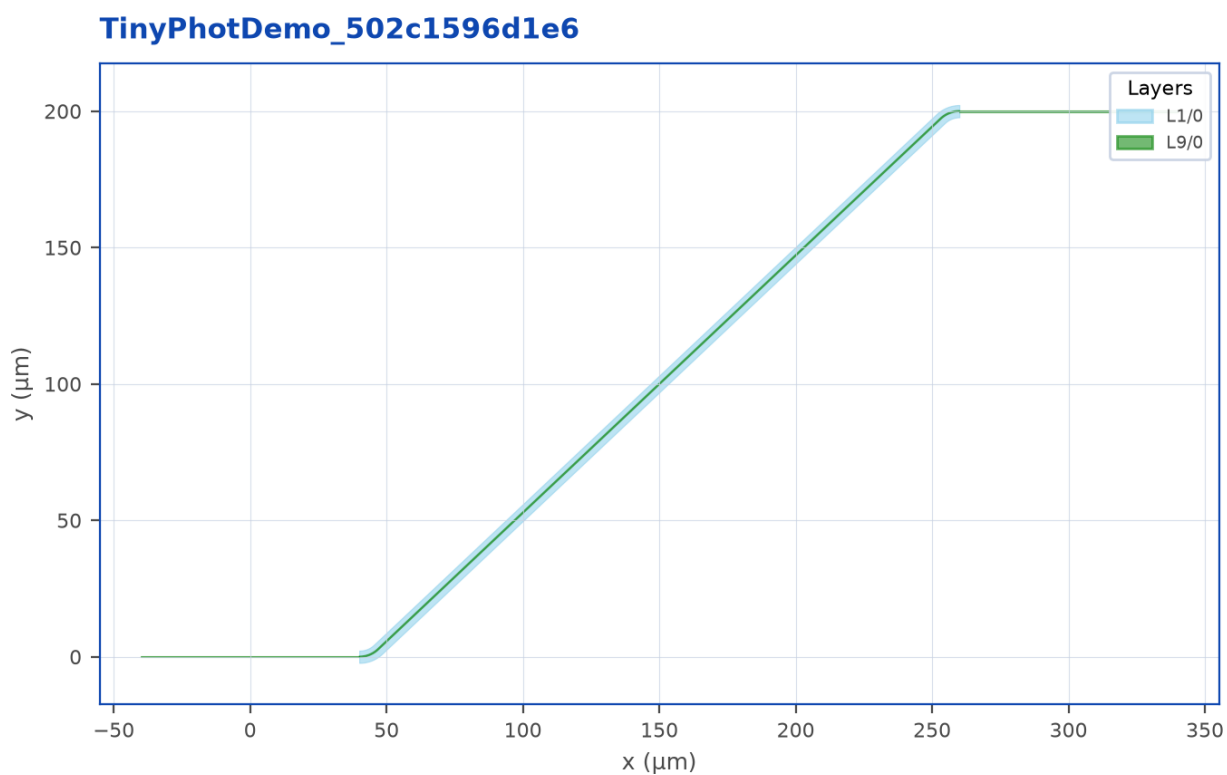


Figure 10.1.: Output of 09_full_minimal_pdk.py - two waveguide stubs joined by an auto-route.

10.1 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.

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

tinyphot/meta.py

```
META = {
    "name": "TinyPhot Research PDK",
    "package-name": "tinyphot",
    "version": "0.1.0",
}
```

That's the conventional shape end users see. The same package source ships in your PyPI wheel; lumicron_pdk.packaging writes the package's Python files at the .lumpdk archive root, and the app materializes them in a private cache for the active project. Relative imports keep the package usable in both environments.

10.2 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-to-MTL1 ohmic contact, SILN-to-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 tested examples, component docstrings, and foundry constraints.

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

10.3 Verifying it works

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

```
python -c "  
import lumicron 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_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.

CHAPTER A

Manifest schema

Every `.lumpdk` contains `manifest.json` at its root. The supported producer is `lumicon_pdk`. packaging; it generates the manifest by importing the PDK package. Do not use the older flat schema with top-level `name`, `module`, or a `layers` array.

A.1 Top-level sections

Field	Type	Producer	Purpose
<code>meta</code>	object	<code><package>.meta.META</code> , or package fallback	Display/import identity and release metadata
<code>layers</code>	object	<code>root LAYER.to_ manifest()</code>	Named GDS layers, colors, and basic rules
<code>stack</code>	array	<code>root STACK</code>	Ordered physical layer stack
<code>route_profiles</code>	object	<code>root RP</code>	Profile port layer, width/radius, and rendered strokes
<code>simulation</code>	object	derived from <code>STACK + RP</code>	Versioned materi- als/extrusions/profile simulation contract
<code>component_schema_ version</code>	integer	<code>packer</code>	Current packaged component contract version: 1

Field	Type	Producer	Purpose
components	array	decorated factories in <package>. components, or package root	Typed parameters, resolved ports, schematic eligibility, circuit models, and descriptions

meta, component_schema_version, and components are generated; other sections depend on what the package exports. Lumicron consumes the manifest for PDK inspection, layer rendering, schematic component fields and ports, and circuit-model selection. A usable app PDK must declare nonempty layers and valid display name, package name, and version.

A.2 meta

Recommended minimum:

```
{
  "meta": {
    "name": "TinyPhot Research PDK",
    "package-name": "tinyphot",
    "version": "0.1.0"
  }
}
```

Field	Type	Current behavior
name	string	Human display name. May contain spaces. Falls back to the package name if no META exists.
package-name	string	Canonical Python import/extraction identity. Must be a valid non-keyword identifier. The packer defaults it to the source package directory name.

Field	Type	Current behavior
package_name	string	Rejected by the current packer. An older manifest may be read for compatibility, but new packages must use package-name.
version	string	Display/release version; falls back to 1.0.0.
other keys	JSON values	Copied through by the packer for PDK-specific metadata. Current app surfaces may ignore them.

package-name and name deliberately serve different purposes. The app requires a declared import identity; it does not derive one from the display name. Use the packer and declare both names explicitly.

A.3 layers

layers is an object keyed by the Python LayerTable name:

```
{
  "layers": {
    "SILC": {
      "layer": 9,
      "datatype": 0,
      "color": "#008000",
      "min_width": 0.2,
      "min_spacing": 0.2,
      "min_radius": 5.0
    },
    "MTL1": {
      "layer": 4,
      "datatype": 0,
      "color": "#ffd700"
    }
  }
}
```

Field	Type	Required from LayerTable	Description
layer	integer	yes	GDS layer number

Field	Type	Required from LayerTable	Description
datatype	integer	yes	GDS datatype
color	string	no	#RRGGBB viewer color
min_width	number	no	Minimum width in micrometers
min_spacing	number	no	Minimum spacing in micrometers
min_radius	number	no	Minimum bend radius in micrometers
display	object	no	Generated fill, stipple and border appearance; no physical mask meaning

Generate this object from LAYER; do not maintain a second hand-written layer list.

display contains optional fill_color and frame_color hex colors, opacity, and either a named pattern or eight pattern_rows. Its optional border object contains width_px, opacity, and style (solid, dashed, dotted, dash-dot, none, or custom). Custom borders additionally contain dashes_px. These lengths are drawable pixels. Omitted border fields preserve the one-pixel opaque solid outline. Older readers may ignore the new border options. Author these fields with `lpdk.Stipple`, `lpdk.Border`, and `lpdk.Layer`; see Chapter 2 for validation bounds.

A.4 stack

Each root STACK entry becomes an ordered object with name, GDS layer/datatype, thickness, z_start, and sidewall_angle_deg. Material index is encoded as constant n, a sellmeier object containing B and C arrays, or a tabulated descriptor with the declared samples. An unspecified material is not an inferred refractive-index model.

```
{
  "stack": [
    {
      "name": "core_si",
      "layer": 9,
      "datatype": 0,
      "thickness": 0.22,
      "z_start": 2.0,
```

```

        "sidewall_angle_deg": 90.0,
        "n": 3.48
    }
]
}

```

A.5 route_profiles

Each named profile records its port layer/datatype, port width, default radius, optional minimum radius, and the concrete continuous strokes returned by the profile. Where declared, bend identifies the default bend and propagation_model carries the circuit propagation descriptor. A profile's routing radius is not evidence that an arbitrary bend PCell is circular or has a known physical radius. For example:

```

{
  "route_profiles": {
    "silicon_strip": {
      "layer": 9,
      "datatype": 0,
      "width": 0.5,
      "radius": 10.0,
      "radius_min": 5.0,
      "strokes": [
        {"layer": 9, "datatype": 0, "width": 0.5, "offset": 0.0}
      ]
    }
  }
}

```

A.6 simulation

The packer derives schema version 1 from STACK and RP. It contains:

- units and reference_wavelength_um;
- process defaults such as background/cladding and default sidewall angle;
- materials keyed by stack-layer name;
- extrusions connecting GDS layers to material and z extents;
- simulation-facing route-profile strokes.

This is technology metadata, not evidence that an electromagnetic solver has run. The supported circuit execution path is SAX; Maple execution remains disabled in this alpha. Keep tabulated material and propagation data tied to their original wavelength range and units.

A.7 components

The packer records each decorated factory's name, module, qualname, schematic flag, symbol, ordered parameters (name, type, required status, and default), and optional description. The legacy params dictionary is retained for older consumers; new consumers use parameters. Layer defaults preserve the GDS pair and, when available, the declared layer name.

ports contains the actual resolved names, directions, cardinal sides, widths, kinds, layer names/GDS pairs, and attached route-profile names. The packer evaluates the default factory and up to 16 boolean parameter variants. Ports controlled by one boolean receive a when condition. Missing required arguments or unsupported conditional metadata produce a port_resolution_error; a component that fails resolution must not be treated as a fully described schematic component.

circuit_models contains the cell's named model declarations, and default_circuit_model names its explicit default. Model declarations retain their engine, kind, reference, settings, port mapping, and source factory. Package-owned Python references are relocated to the installed lumicon.pdks.<package-name> namespace; declared data references remain package-relative and their files are included. Model-resolution failures are recorded in model_resolution_error. The packer does not infer a simulation model from a component's name or geometry.

gds_sources, when present, records imported layout resource paths, content hashes, and top cells. The corresponding GDS assets are packaged and checked for changes before publication.

A.8 Compatibility and validation boundary

- The app requires a valid ZIP with root manifest.json, the required metadata, and nonempty layers. The current packer emits component_schema_version: 1.
- Invalid package-name values fail packing. The output filename must be <package-name>.lumpdk; release versions belong in metadata or parent directories.
- Archive entries must use safe relative paths. Code and referenced resources are materialized in a private cache, rather than a user-maintained global UserPDKs folder.
- The packer includes Python source, declared compact-model data, declared optical CSV sources, package-local GDS assets, and optional drc.json. It does not copy arbitrary documentation, EULAs, or unrelated assets merely because they are mentioned in metadata.
- Current limits include 4 MiB for the serialized manifest and 128 MiB for the archive. Optical CSV source files are limited to 16 MiB each. Oversized samples are not silently downsampled.
- lumicon_required is not an enforced compatibility gate. Test the package against the target app/API build and publish that identity in release notes.

Use the packer, inspect the resulting ZIP, load it with **Load PDK**, and run a clean canonical import and representative circuit/layout examples as the release gate.

CHAPTER B

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 `lpdk.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 - generated `manifest.json` inside a `.lumpdk` archive. Carries nested meta (display name, importable package-name, version) plus layers, stack, profiles, simulation, and component meta-data when available.

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 @lpdk.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) to a 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 with root manifest.json, package-relative Python source, and optional drc.json.

API reference

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

lpgk.Border

Independent layer outline color, opacity, width and dash style.

```
lpgk.Border(*, width_px: 'float' = 1, color: 'str | None' = None, style: 'str | None' =  
↳ None, opacity: 'float' = 1, dashes_px: 'Sequence[float] | None' = None)
```

lpgk.CompactModel

Reusable immutable model description for a custom component library.

```
lpgk.CompactModel(reference, *, settings=None, ports=None, data_options=None,  
↳ technology_context=None)
```

lpgk.Layer

A GDS layer with optional design rules and display color.

```
lpgk.Layer(layer_num: 'int', datatype: 'int' = 0, *, layer: Annotated[int, Ge(ge=0),  
↳ Le(le=65535)], color: str | None = None, min_width: float | None = None,  
↳ min_spacing: float | None = None, min_radius: float | None = None, stipple:  
↳ lumicron._technology.display.Stipple | None = None, opacity: float | None = None,  
↳ border: lumicron._technology.display.Border | None = None) -> None
```

lpmk.LayerMap

Named mapping of layer names to LayerSpec values.

```
lpmk.LayerMap(*, layers: dict[str, lumicron._technology.layer.LayerSpec] = <factory>)
↳ -> None
```

lpmk.LayerStack

Ordered physical layer stack for simulation and virtual fabrication.

```
lpmk.LayerStack(*stack_layers: 'StackLayer')
```

lpmk.LayerTable

Named collection of Layer objects with attribute access.

```
lpmk.LayerTable(**layers: 'Layer')
```

lpmk.PhotonicElevator

Create a two-layer photonic elevator cell (inverse-taper transition).

```
lpmk.PhotonicElevator(layer_a: 'LayerSpec', layer_b: 'LayerSpec', width_a: 'float' =
↳ 0.5, width_b: 'float' = 0.5, tip_a: 'float' = 0.08, tip_b: 'float' = 0.08,
↳ taper_length_a: 'float' = 50.0, taper_length_b: 'float' = 50.0, overlap: 'float' =
↳ 10.0, cladding_layer: 'LayerSpec | None' = None, cladding_margin: 'float' = 2.0,
↳ name: 'str | None' = None, port_type: 'str' = 'optical', route_profile_a=None,
↳ route_profile_b=None)
```

lpmk.ProfileLayer

One cross-section stroke emitted by a route profile.

```
lpmk.ProfileLayer(*, layer: lumicron._technology.layer.LayerSpec, width:
↳ Annotated[float, Gt(gt=0)], offset: float = 0.0) -> None
```

lpmk.ProfileTooth

One periodic PCell instance tiled along the route centerline.

```
lpmk.ProfileTooth(cell: 'Any', period: 'float', offset: 'float' = 0.0, phase:
↳ 'Optional[float]' = None, span_bends: 'bool' = True) -> None
```

lpmk.RouteProfile

Route profile — produces one or more parallel paths per route.

```
lpmk.RouteProfile(name: 'str', port_layer: 'LayerSpec', port_width: 'float', radius:
↳ 'float' = 0.0, radius_min: 'float' = 0.0, builder: 'Optional[Callable]' = None,
↳ bend: 'Any' = None, propagation: 'Any' = None, routing_policy: 'RoutingPolicy |
↳ None' = None)
```

lpmk.RouteProfiles

Named collection of RouteProfile objects with attribute access.

```
lpmk.RouteProfiles(**profiles: 'RouteProfile')
```

lpmk.RoutingPolicy

PDK permission for generic width-transition geometry.

```
lpmk.RoutingPolicy(auto_taper: 'bool' = True) -> None
```

lpmk.Sellmeier

Wavelength-dependent refractive index via Sellmeier equation.

```
lpmk.Sellmeier(*, B: list[float], C: list[float]) -> None
```

IpdK.StackLayer

A single layer in the physical stack.

```
lpadk.StackLayer(name: 'str', *, layer: lumicon._technology.layer.LayerSpec,
  ↳ thickness: Annotated[float, Gt(gt=0)], z_start: float = 0.0, n: float | None =
  ↳ None, sellmeier: lumicon._technology.stack.Sellmeier | None = None, tabulated:
  ↳ lumicon._technology.dispersion.TabulatedIndex | None = None, sidewall_angle_deg:
  ↳ Annotated[float, Gt(gt=0.0), Lt(lt=180.0)] = 90.0) -> None
```

IpdK.Stipple

An immutable 8×8 fill mask; use named constants or Stipple(rows=...).

```
lpadk.Stipple(*, rows: 'Sequence[str]')
```

IpdK.TabulatedIndex

Immutable positive real index samples with bounded PCHIP interpolation.

```
lpgk.TabulatedIndex(*, schema_version: Literal[1] = 1, interpolation: Literal['pchip']
↳ = 'pchip', role: Literal['bulk', 'effective'], wavelengths_um:
↳ Annotated[tuple[Annotated[float, FieldInfo(annotation=NoneType, required=True,
↳ metadata=[Strict(strict=True), Gt(gt=0),
↳ _PydanticGeneralMetadata(allow_inf_nan=False)]]], ...], MinLen(min_length=2),
↳ MaxLen(max_length=100000)], indices: Annotated[tuple[Annotated[float,
↳ FieldInfo(annotation=NoneType, required=True, metadata=[Strict(strict=True),
↳ Gt(gt=0), _PydanticGeneralMetadata(allow_inf_nan=False)]]], ...],
↳ MinLen(min_length=2), MaxLen(max_length=100000)], source_name:
↳ Optional[Annotated[str, StringConstraints(strip_whitespace=True, to_upper=None,
↳ to_lower=None, strict=None, min_length=1, max_length=256, pattern=None,
↳ ascii_only=None)]] = None, source_sha256: Optional[Annotated[str,
↳ FieldInfo(annotation=NoneType, required=True,
↳ metadata=[_PydanticGeneralMetadata(pattern='^[0-9a-f]{64}$')]]]] = None,
↳ polarization: Optional[Annotated[str, StringConstraints(strip_whitespace=True,
↳ to_upper=None, to_lower=None, strict=None, min_length=1, max_length=256,
↳ pattern=None, ascii_only=None)]] = None, mode: Optional[Annotated[str,
↳ StringConstraints(strip_whitespace=True, to_upper=None, to_lower=None,
↳ strict=None, min_length=1, max_length=256, pattern=None, ascii_only=None)]] =
↳ None, temperature_k: Optional[Annotated[float, FieldInfo(annotation=NoneType,
↳ required=True, metadata=[Strict(strict=True), Gt(gt=0),
↳ _PydanticGeneralMetadata(allow_inf_nan=False)]]]] = None, cross_section:
↳ Optional[Annotated[str, StringConstraints(strip_whitespace=True, to_upper=None,
↳ to_lower=None, strict=None, min_length=1, max_length=256, pattern=None,
↳ ascii_only=None)]] = None) -> None
```

lpgk.TransitionRegistry

Maps (layer_a, layer_b) → transition factory.

```
lpgk.TransitionRegistry() -> 'None'
```

lpgk.build_manifest

Import a PDK package and build its manifest dict.

```
lpgk.build_manifest(pdk_module_name: 'str') -> 'dict'
```

lpgk.default_transitions

Return the process-wide default transition registry.

```
lpdk.default_transitions() -> 'TransitionRegistry'
```

lpdk.describe_pdk

Introspect a PDK module and return its layer table, physical stack, PCells, route profiles, and module-level constants.

```
lpdk.describe_pdk(pdk_name: 'str') -> 'dict'
```

lpdk.open_pdk_authoring_guide

Open the PDK Authoring Guide PDF in the system's default viewer.

```
lpdk.open_pdk_authoring_guide() -> pathlib.Path
```

lpdk.pack_pdk

Pack a PDK Python package into a .lumpdk archive.

```
lpdk.pack_pdk(pdk_module_name: 'str', output: 'str | Path | None' = None, *,  
↳ output_directory: 'str | Path | None' = None) -> 'Path'
```

lpdk.pdk_authoring_guide_path

Filesystem path to the bundled PDK Authoring Guide PDF.

```
lpdk.pdk_authoring_guide_path() -> pathlib.Path
```

lpdk.route_profile

Decorator that turns a cross-section builder function into a RouteProfile.

```
lpmk.route_profile(*, name: 'Optional[str]' = None, port_layer: 'Optional[LayerSpec]'
↳ = None, port_width: 'float', radius: 'float' = 0.0, radius_min: 'Optional[float]'
↳ = None, bend: 'Any' = None, propagation: 'Any' = None, routing_policy:
↳ 'RoutingPolicy | None' = None) -> 'Callable'
```

lpmk.tests

Standard PDK checks and inspection artifacts with explicit evidence bounds.

lpmk.tests.CheckResult

CheckResult(check: str, status: str, reason: str, subject: str | None = None, artifacts: tuple[str, ...] = (), warnings: tuple[str, ...] = ())

```
lpmk.tests.CheckResult(check: str, status: str, reason: str, subject: str | None =
↳ None, artifacts: tuple[str, ...] = (), warnings: tuple[str, ...] = ()) -> None
```

lpmk.tests.TestReport

Bound results, never a blanket ‘fully qualified’ PDK certificate.

```
lpmk.tests.TestReport(package: str, input_digest: str, input_files: dict,
↳ runtime_digest: str, output_directory: str, checks:
↳ tuple[lumicron_pdk.tests.CheckResult, ...], artifact_digests: dict,
↳ schema_version: int = 1, owner: str = 'lumicron_pdk.tests/v1',
↳ environment_binding: dict | None = None) -> None
```

lpmk.tests.available

Discover the standard check groups supported by this runtime.

```
lpmk.tests.available() -> tuple[str, ...]
```

lpdk.tests.run

Run standard checks in fresh imports and replace only owned output.

```
lpdk.tests.run(package: str, *, output_directory: str | pathlib.Path, checks:  
    ↳ tuple[str, ...] | None = None, timeout: float = 120, environment=None) ->  
    ↳ lumicron_pdk.tests.TestReport
```

lpdk.Stipple.CHECKERBOARD

An immutable 8×8 fill mask; use named constants or Stipple(rows=...).

lpdk.Stipple.CROSSHATCH

An immutable 8×8 fill mask; use named constants or Stipple(rows=...).

lpdk.Stipple.DENSE_DOTS

An immutable 8×8 fill mask; use named constants or Stipple(rows=...).

lpdk.Stipple.DENSE_GRID

An immutable 8×8 fill mask; use named constants or Stipple(rows=...).

lpdk.Stipple.DENSITY_25

An immutable 8×8 fill mask; use named constants or Stipple(rows=...).

lpdk.Stipple.DENSITY_50

An immutable 8×8 fill mask; use named constants or Stipple(rows=...).

IpdK.Stipple.DENSITY_75

An immutable 8×8 fill mask; use named constants or Stipple(rows=...).

IpdK.Stipple.DIAGONAL_CROSSHATCH

An immutable 8×8 fill mask; use named constants or Stipple(rows=...).

IpdK.Stipple.DIAGONAL_DOWN

An immutable 8×8 fill mask; use named constants or Stipple(rows=...).

IpdK.Stipple.DIAGONAL_UP

An immutable 8×8 fill mask; use named constants or Stipple(rows=...).

IpdK.Stipple.EMPTY

An immutable 8×8 fill mask; use named constants or Stipple(rows=...).

IpdK.Stipple.HORIZONTAL

An immutable 8×8 fill mask; use named constants or Stipple(rows=...).

IpdK.Stipple.MEDIUM_DOTS

An immutable 8×8 fill mask; use named constants or Stipple(rows=...).

IpdK.Stipple.SOLID

An immutable 8×8 fill mask; use named constants or Stipple(rows=...).

lpdk.Stipple.SPARSE_DOTS

An immutable 8×8 fill mask; use named constants or Stipple(rows=...).

lpdk.Stipple.SPARSE_GRID

An immutable 8×8 fill mask; use named constants or Stipple(rows=...).

lpdk.Stipple.VERTICAL

An immutable 8×8 fill mask; use named constants or Stipple(rows=...).

lpdk.Layer.model_post_init

This function is meant to behave like a BaseModel method to initialize private attributes.

```
lpdk.Layer.model_post_init(context: 'Any', /) -> 'None'
```

lpdk.Layer.rules

Print design rules for this layer.

```
lpdk.Layer.rules() -> 'None'
```

lpdk.LayerMap.add

Public API symbol exported by this build.

```
lpdk.LayerMap.add(name: 'str', layer: 'int', datatype: 'int' = 0) -> 'LayerSpec'
```

lpdk.LayerMap.get

Public API symbol exported by this build.

```
lpdk.LayerMap.get(name: 'str') -> 'LayerSpec'
```

lpdk.LayerStack.print_stack

Print the physical layer stack.

```
lpdk.LayerStack.print_stack() -> 'None'
```

lpdk.LayerStack.z_max

Highest physical boundary in micrometres; zero for an empty stack.

```
lpdk.LayerStack.z_max # property
```

lpdk.LayerStack.z_min

Lowest physical boundary in micrometres; zero for an empty stack.

```
lpdk.LayerStack.z_min # property
```

lpdk.LayerTable.print_rules

Print all layers with their design rules.

```
lpdk.LayerTable.print_rules() -> 'None'
```

lpdk.LayerTable.to_layer_map

Generate a LayerMap for serialization.

```
lpdk.LayerTable.to_layer_map() -> 'LayerMap'
```

lpdk.LayerTable.to_manifest

Export layers as a dict suitable for manifest.json.

```
lpdk.LayerTable.to_manifest() -> 'dict[str, dict]'
```

lpdk.RouteProfile.render

Expand the profile builder into concrete ProfileLayer strokes.

```
lpdk.RouteProfile.render(port_width: 'Optional[float]' = None) -> 'list[ProfileLayer]'
```

lpdk.RouteProfile.render_teeth

Expand the profile builder into ProfileTooth specs (may be empty).

```
lpdk.RouteProfile.render_teeth(port_width: 'Optional[float]' = None) ->  
↪ 'list[ProfileTooth]'
```

lpdk.RouteProfile.rules

Print route profile details.

```
lpdk.RouteProfile.rules() -> 'None'
```

lpdk.RouteProfiles.print_rules

Public API symbol exported by this build.

```
lpdk.RouteProfiles.print_rules() -> 'None'
```

lpdk.Sellmeier.n

Compute refractive index at wavelength (in microns).

```
lpgk.Sellmeier.n(wavelength: 'float') -> 'float'
```

lpgk.StackLayer.n_at

Refractive index at the given wavelength (microns).

```
lpgk.StackLayer.n_at(wavelength: 'float' = 1.55) -> 'float | None'
```

lpgk.TabulatedIndex.dn_dlambda

Derivative of the PCHIP index interpolant with respect to wavelength (1/um).

```
lpgk.TabulatedIndex.dn_dlambda(wavelength_um: 'Any') -> 'Any'
```

lpgk.TabulatedIndex.from_csv

Read UTF-8 CSV with a header and at least two complete sample rows.

```
lpgk.TabulatedIndex.from_csv(path: 'str | Path', *, wavelength_column: 'str | None' =
  ↳ None, index_column: 'str | None' = None, wavelength_unit: 'str | None' = None,
  ↳ role: "Literal['bulk', 'effective'] | None" = None) -> 'TabulatedIndex'
```

lpgk.TabulatedIndex.group_index

Return $n - \lambda \, dn/d\lambda$ from the same interpolant, preserving scalar/array shape.

```
lpgk.TabulatedIndex.group_index(wavelength_um: 'Any') -> 'Any'
```

lpgk.TabulatedIndex.model_post_init

This function is meant to behave like a BaseModel method to initialize private attributes.

```
lpgk.TabulatedIndex.model_post_init(context: 'Any', /) -> 'None'
```

lpmk.TabulatedIndex.n

Index at scalar or array wavelengths in um, within the sampled range.

```
lpmk.TabulatedIndex.n(wavelength_um: 'Any') -> 'Any'
```

lpmk.TabulatedIndex.wavelength_range_um

Inclusive measured wavelength range, in micrometres.

```
lpmk.TabulatedIndex.wavelength_range_um # property
```

lpmk.TransitionRegistry.lookup

Return (factory, reversed) for the layer pair, or None.

```
lpmk.TransitionRegistry.lookup(layer_a: 'LayerSpec', layer_b: 'LayerSpec') ->  
  ↳ 'tuple[Callable, bool] | None'
```

lpmk.TransitionRegistry.register

Register *factory* as the transition from layer_a to layer_b.

```
lpmk.TransitionRegistry.register(layer_a: 'LayerSpec', layer_b: 'LayerSpec', factory:  
  ↳ 'Callable', *, symmetric: 'bool' = True) -> 'None'
```

lpmk.tests.TestReport.artifacts_intact

Public API symbol exported by this build.

```
lpmk.tests.TestReport.artifacts_intact() -> bool
```

lpdk.tests.TestReport.counts

Public API symbol exported by this build.

```
lpdk.tests.TestReport.counts # property
```

lpdk.tests.TestReport.is_current

Exact source/assets and API implementation binding, not authenticity.

```
lpdk.tests.TestReport.is_current(package_directory: str | pathlib.Path, *,  
    ↪ environment=None) -> bool
```

lpdk.tests.TestReport.summary

Public API symbol exported by this build.

```
lpdk.tests.TestReport.summary() -> str
```

lpdk.tests.TestReport.to_dict

Public API symbol exported by this build.

```
lpdk.tests.TestReport.to_dict() -> dict
```