# 3-ID-C BITS

A tutorial walkthrough

*Self-paced; ~30 slides; ~45-60 minutes*

*Presented 2026-06-01*

## Audience

This deck is for someone who:

- Knows the 3-ID-C hardware and beamline workflow
- Has used SPEC or bare EPICS (`caget` / `caput` /MEDM)
- Has *some* Python familiarity (knows what a dict and a function are; not necessarily comfortable writing classes)
- Has heard "Bluesky is replacing SPEC" and wants to know what that actually means in practice

If you've already seen the standard intro deck, this one fills in the *why*.

## What we'll cover

1. The shape of a BITS session
2. SPEC -> Bluesky: command cross-walk
3. EPICS -> ophyd: the device model
4. Plans, plan stubs, and `yield from`
5. The RunEngine: pause/resume, suspenders, document streams
6. What's in the 3-ID-C instrument today
7. The omega <-> laser_optics interlock (design + scope)
8. Inspecting past data via Tiled
9. The docs site and how to extend it
10. Where Bluesky is worse than SPEC (honest section)

# Part 1: shape of a session

```
conda activate 3idc-bits
ipython
```

```
from id3c.startup import *
```

That `from ... import *` triggers:

- Load `iconfig.yml` (top-level session config)
- Create the **RunEngine** (`RE`) with `bec` and `tiled` subscribers
- Instantiate every device in `configs/devices.yml`
- Install the omega <-> laser_optics interlock
- Register IPython magics (`%wa`, `%mov`, `%ct`)
- Import demo plans (`sim_print_plan`, ...)

## Part 1: what's bound at the prompt (1/2) -- machinery

```
RE                          # the RunEngine
oregistry                   # the device registry
cat                         # the Tiled catalog client
bec                         # BestEffortCallback (live plots/tables)

bps                         # bluesky.plan_stubs (mv, sleep, ...)
bp                          # bluesky.plans (count, scan, ...)
```

## Part 1: what's bound at the prompt (2/2) -- devices

```
sample_stage              # a MotorBundle
laser_optics              # our custom LaserOptics
shutter                   # ApsPssShutter
eiger2                    # area detector
sim_motor, sim_det        # simulators
```

`%wa` shows everything. Tab-completion works on devices.

`listobjects()` Concise table of all top-level control devices.

# Part 2: SPEC -> Bluesky (1/2) -- motion and inspection

| SPEC | Bluesky |
|---|---|
| `mv samx 5` | `RE(bps.mv(sample_stage.xprime, 5))` |
| `mvr samx 0.1` | `RE(bps.mvr(sample_stage.xprime, 0.1))` |
| `wm samx` | `sample_stage.xprime.position` (no `RE`!) |
| `wa` | `%wa` |
| `shopen` | `RE(bps.mv(shutter, "open"))` |

# Part 2: SPEC -> Bluesky (2/2) -- counts and scans

| SPEC | Bluesky |
| --- | --- |
| `ct 1` | `RE(bp.count([scaler]))` |
| `ascan samx 0 10 10 1` | `RE(bp.scan([scaler], sample_stage.xprime, 0, 10, 11))` |
| `mesh ...` | `RE(bp.grid_scan([scaler], mot1, ..., mot2, ...))` |

**Note SPEC's "10 intervals" vs. Bluesky's "11 points".**

## Part 2: things SPEC does that Bluesky doesn't

Being honest:

- **Compactness.** `ascan samx 0 10 10 1` is shorter than the Bluesky equivalent. We can alias common commands, but bare commands are longer.
- **Macros.** `overnight.mac` is faster to write than authoring a Python plan.
- **One command -> one file.** SPEC files are human-readable text. Bluesky runs live in a Tiled catalog; you need a client to read.
- **Decades of stability.** SPEC's command set is unchanged; Bluesky is younger.
- **Programming Language** -- SPEC's uses a familiar C-like syntax but is unique to SPEC, the language support is limited to the SPEC user community.

## Part 2: things Bluesky does that SPEC doesn't (1/2)

- **Structured metadata** -- every run has a UID, a `scan_id`, and a `md=` dict you can search later.
- **Pause / resume.** `Ctrl-C Ctrl-C` pauses mid-scan; `RE.resume()` continues.
- **Document streams** -- `start`, `descriptor`, `event`, `stop` documents flow to *subscribers* (BEC for live plots, TiledWriter for storage). Standardized format.
- **Suspenders.** "Pause on beam dump, resume when it returns" is a generic mechanism, not bespoke per-beamline code.

## Part 2: things Bluesky does that SPEC doesn't (2/2)

- **Programming Language** -- Python is a popular & well-documented language. Help is available from many sources.
- **Syntax Checking** -- Syntax checking is inherent to Python. SPEC macros are string text, interpreted at run time.
- **Catalog-backed history.** `cat[-1]` is the most recent run, `cat[uid]` is a specific one; you can search by metadata.
- **Area detectors.** ophyd wraps the full EPICS areaDetector framework: the cam, the plugin chain (ROI, stats, codecs, file writers, PVA push, ...), and per-run orchestration of all of it. HDF5-via-external-links is one example; see how_to/visualize_hdf5.

## Part 3: EPICS -> ophyd

You used to type the PV string:

```
caget 3idxps1:m5.RBV
caput 3idxps1:m5.VAL 30
```

Now you address a Python object that wraps the PVs:

```
sample_stage.omega.user_readback.get()    # the .RBV
sample_stage.omega.user_setpoint.put(30)  # the .VAL
sample_stage.omega.move(30)               # set + wait
```

An `EpicsMotor` wraps ~12 PVs (.VAL, .RBV, .DMOV, .MOVN, .STOP, .HLM, .LLM, .EGU, .OFF, ...). You access them as attributes.

## Part 3: why wrap PVs?

The trade:

- You **lose** "any PV at any time" -- you have to define what signals a device has.
- You **gain**:
  - Self-documenting object (tab-completion!)
  - Search PV and connect once.
  - Long-lived connection with cached reads
  - Subscription-first API
  - `read()` -> structured dict ready for archiving
  - Integration with the Bluesky document stream

For *one* CA operation, `caget` is still fine. For an **instrument**, the wrapper pays for itself many times over.

## Part 3: `get()` vs `read()`

Two operations users confuse:

```
sample_stage.omega.user_readback.get()
# 30.0           -- ONE signal's value

sample_stage.omega.read()
# {'sample_stage_omega':              {'value': 30.0, 'timestamp': ...},
#  'sample_stage_omega_user_setpoint': {'value': 30.0, 'timestamp': ...}}
# -- ALL signals of kind hinted/normal
```

`get()` for "give me a number." `read()` for "give me a snapshot."

`device.read()` is what the RunEngine calls internally during a scan.

## Part 4: plans and plan stubs

A **plan** publishes Bluesky documents: examples `bp.count`, `bp.scan`
A **plan stub** does not publish: examples `bps.mv`, `bps.sleep`

Both are *generators*: functions that yield messages for the RunEngine. Both work with `RE(...)`.

The difference matters when you *write* plans:

- Plan stubs are easy: yield from other stubs, do not bracket a run.
- Plans need `open_run` / `close_run` (or compose a `bp.*`).

Most user code is plan stubs. Composing into a plan is usually wrapping an existing `bp.*` plan and including calls to plan stubs as needed.

## Part 4: `yield from`

`yield from` is the Python syntax for **composing one generator inside another**.

```
@plan
def my_plan():
    yield from bps.mv(motor, 5)         # composed stub
    yield from bp.count([detector])     # composed plan
```

You use it **inside** a plan you are writing. You do not use it at the IPython prompt. At the prompt, use `RE(...)`.

If you forget either, the generator is created and discarded -- the call does nothing. The `@plan` decorator catches that.

## Part 4: the `@plan` decorator

All plans and plan stubs we author are decorated with `bluesky.utils.plan`. If you call one without `RE(...)`, a warning prints shortly after you press Enter:

```
RuntimeWarning: plan `sim_print_plan` was never iterated,
                did you mean to use `yield from`?
```

The warning's traceback points at *your* command line, not at internal Bluesky code, so you can see exactly which line to retype.

Convention: every new plan/plan-stub in `src/id3c/` gets `@plan`.
See `AGENTS.md` > "`@plan` decorator on our own plans".

## Part 5: the RunEngine

The RunEngine is the thing that **executes** a plan. It:

- Iterates the generator one message at a time
- Dispatches each message to the appropriate device
- Publishes documents to subscribers (BEC, TiledWriter)
- Receives CA monitors and caches values
- Handles pauses, suspenders, errors, cleanup
- Threads metadata through the document stream

You can think of it as the SPEC interpreter, but for plans.
The RunEngine is *the* thing that turns a description of a scan into an actual scan.

## Part 5: pause / resume

During a long scan:

- `Ctrl-C` once -- deferred pause (after current message finishes)
- `Ctrl-C Ctrl-C` -- immediate pause (interrupts the current await)

At the pause prompt:

```
RE.resume()    # continue
RE.abort()     # finish, exit_status='abort'

# used less often
RE.stop()      # finish cleanly, success
RE.halt()      # emergency stop, no documents
```

This is **free** -- works for every plan including custom ones.

# Part 5: subscribers and the document stream

Every `RE(plan)` invocation emits a stream of documents:

| document type | when |
|---|---|
| `start` | once, at plan begin |
| `descriptor` | once per data stream |
| `event` | once per data point |
| `stop` | once, at plan end |

Subscribers consume the stream live:

- **`bec`** -- BestEffortCallback: prints tables, opens plots
- **TiledWriter** -- sends documents to the Tiled server
- **nxwriter** (optional) -- writes NeXus-format HDF5 files

You can add your own: `RE.subscribe(my_callback)`.

## Part 6: what's installed today

| device | notes |
|---|---|
| `sample_stage` | xprime / base_y / zprime / **omega** (interlocked) |
| `detector_stage` | det_x / eiger_y / eiger_z |
| `laser_optics` | us / ds (interlocked with omega) |
| `shutter` | A-station PSS shutter |
| `eiger2` | Eiger2 500k; HDF5 file plugin still FIXME |
| `sim_motor`, `sim_det` | simulators for verification |

`%wa` lists everything by label; `%wa baseline` shows the devices recorded at the start and end of every run.

## Part 7: the omega <-> laser_optics interlock

**Why:** when the laser optics are not retracted, they could collide with the rotating sample stage; symmetrically, pulling the laser in/out while omega is moving is also risky.

**What is protected:**

- `sample_stage.omega` blocked unless `laser_optics.is_out` (both axes within +/- 1 mm of -75 mm)
- `laser_optics.us` / `.ds` blocked while `omega.motor_is_moving`

**How:** `InterlockedEpicsMotor.move()` runs an interlock check before any CA put (pre-flight) **and** subscribes a watcher to the relevant signals during motion (mid-flight). Failure raises `MotionInterlock`.

## Part 7: scope of the interlock

**This is a Python-session interlock.** It does not:

- Write to EPICS PV disable fields
- Install IOC sequencer code
- Protect against MEDM jogs
- Protect against `caput` from a shell
- Protect against a different Bluesky session
- Survive a Python process crash

For *session-independent* hardware-grade protection, the right place is the IOC (CALC/SCALC, state notation, or a soft record driving PV disable). Adding that is a separate (welcome) project.

This is documented honestly in the module docstring of `interlocked_motor.py` and in `docs/source/explanation/interlocks.md`.

## Part 8: inspecting past data

Bluesky runs are written to a Tiled server (3-ID-C uses http://sn.xray.aps.anl.gov:8000). The session-level client is `cat`:

```
cat[-1]                          # most recent run
cat["<uid>"]                     # by UID

run = cat[-1]
run.metadata["start"]            # plan args, scan_id, plan_name, ...
run.primary.read()               # xarray Dataset of the main stream
run.baseline.read()              # baseline-labeled devices
```

## Part 8: image data and HDF5 external links

For an area-detector run, the data flow is:

```
Eiger IOC -> writes image.h5
custom Bluesky plan  -> writes master.h5 with HDF5 external link to image.h5
TiledWriter -> sends run docs referencing master.h5
client reads: client -> Tiled -> master.h5 -> image.h5 (via link)
```

The chain works *if* every hop succeeds, especially the last (image file visible to the Tiled server). At 3-ID-C, this end-to-end path is **not yet validated**. See `docs/source/how_to/visualize_hdf5.md` for the current state.

## Part 9: the docs site

`docs/source/` is a Sphinx site, Diátaxis-organized:

- **tutorials/** -- *learning* (first session, SPEC->Bluesky, EPICS->ophyd)
- **how_to/** -- *task* (add a device, add a plan, inspect data, ...)
- **reference/** -- *lookup* (cheat sheet, quick reference, configuration)
- **explanation/** -- *understanding* (RunEngine, plans+stubs, interlocks)

Plus this presentations directory.

Build locally: `cd docs && make html`. CI deploys `main` to `https://bcda-aps.github.io/3idc-bits/`.

## Part 9: how to extend the docs

| Add a... | ...where |
|---|---|
| New page | `docs/source/<section>/<name>.md` + toctree entry |
| New device | `src/id3c/configs/devices.yml` (or a class in `src/id3c/devices/`) |
| Custom motor | `mb_creator` per-axis `class:` key in YAML |
| Plan | `src/id3c/plans/<topic>.py` + `startup.py` import |
| Interlock | `src/id3c/devices/<a>_<b>_interlock.py` + `startup.py` line |
| Subscriber callback | `src/id3c/callbacks/<name>.py` + `RE.subscribe` |

Every category has a matching how-to page; the `reference/quick_reference.md` table has the full mapping.

# Part 10: honest summary

**Bluesky gains** (vs SPEC):

- Reproducibility, recoverability, post-experiment data access
- Pause / resume, suspenders, structured metadata
- Live plots and tables for free
- A real software stack you can hire Python developers for

**Bluesky pains** (vs SPEC):

- More verbose syntax
- A learning curve (this deck exists for a reason)
- More layers between you and the PVs when things go wrong
- Tracebacks are long; learn to read the bottom line

A worthy trade to gain recoverable data and reproducible workflows.

## Where to go from here

- Open IPython. Run `from id3c.startup import *`. Type `%wa`.
- Run the three sim plans: `RE(sim_print_plan())`, `RE(sim_count_plan())`, `RE(sim_rel_scan_plan())`. Watch BEC plot the third one.
- Try forgetting `RE(...)` once on purpose: `sim_print_plan()`. See the `RuntimeWarning`. Internalize it.
- Read the cheat sheet. Print it. Tape it next to your monitor.
- File issues for anything that confuses you.

Welcome to Bluesky at 3-ID-C.

## References

- This repo: https://github.com/BCDA-APS/3idc-bits
- Docs: https://bcda-aps.github.io/3idc-bits/
- Bluesky upstream: https://blueskyproject.io/
- apsbits: https://github.com/BCDA-APS/apsbits
- Tiled: https://blueskyproject.io/tiled/
- Diátaxis (doc structure): https://diataxis.fr/
- **Bluesky Office Hours:** Every Wednesday, 2-3 pm on Teams

Questions: https://github.com/BCDA-APS/3idc-bits/issues