ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WEPHA083

`ophyd` DEVICES: IMPOSING HIERARCHY ON THE FLAT EPICS V3 NAMESPACE *

K. Lauer†, SLAC National Accelerator Laboratory, Menlo Park,USA

Abstract

EPICS V3 provides simple data types accessible over the network through Channel Access identified by a flat process variable (PV) name. This flexibility is often regarded as a strength of EPICS, as the user can easily pick and choose the information they require. However, such data is almost always inter-related in some manner, pushing the burden of reconstructing that relationship to the end-user/client.

ophyd represents hardware in Python as hierarchical classes, grouping together related signals from the underlying control system. ophyd devices make imposing this hierarchy simple, readable, and descriptive. This structure allows ophyd to provide a consistent interface across a wide-range of devices, which can then be used by higher-level software for any number of tasks: from command-line inspection, to scanning/data collection (bluesky), or even automatic GUI generation (typhon, adviewer). ophyd contains a number of pre-built devices for common hardware (and IOCs) as well as the tools to build custom devices.

BACKGROUND

EPICS and PVs

Standard EPICS IOCs host a process database of records. Records, which generally hold a primary value in the field `.VAL` along with related metadata in other appropriately-named fields (e.g., `.DESC` for description, `.EGU` for engineering units, etc.), are made available to clients and other servers over the Channel Access (CA) protocol. In a properly configured network of IOCs, records are almost always uniquely named such that only one IOC on one machine serves information from its database. At that point, a specific field of a record `RECORD.FIELD` over CA is often referred generically to as getting or putting to a Process Variable (PV).

Values along with a fixed set of metadata can be retrieved over CA in a single request. In CA, it is not possible to group together multiple requests and have the server return an atomic result.

A Short Note about PVAccess

Much additional work has been put into the most recent major version of EPICS (V7) in recent years to bring structured data to the protocol level, which is not currently possible in Channel Access, with a new protocol called PVAccess. This addition allows for keeping structured data accessible and synchronized at the IOC server level.

Such synchronization is outside of the scope of ophyd, which currently relies on CA to retrieve or change PVs on IOCs.

While ophyd does not currently have PVAccess support, it is a feature that is currently in the planning stages. The composability, configurability, and consistent API of devices, as described in later sections, will still apply when ophyd is PVAccess-capable – even allowing for CA, PVAccess, and soft/simulation signals to be mixed in as-needed.

There are thousands of deployed EPICS V3 servers that may never see an upgrade to V7 for a variety of reasons, meaning that the relevance of CA and related higher-level applications will likely persist for decades.

SIGNALS

`ophyd.Signal`

An ophyd `Signal` represents the smallest set of data a user might be interested in – a single temperature value, a PID setpoint or readback, and so on. The data held by a `Signal` may be structured and may have additional metadata associated with it, including timestamps and control limits.

Signals can be used in isolation, instantiated as needed. The strength of ophyd comes in when signals are used in conjunction with devices, which is detailed in the next sections.

`ophyd.EpicsSignal`

A subclass of `ophyd.Signal`, an `EpicsSignal` bridges the gap between CA and the ophyd signal interface.

As setpoints and readback values are often separate PVs in EPICS, an `EpicsSignal` allows for specifying a PV to write to (setpoint) and a PV from which to read (readback).

A simple example might be that of the motor record[1], where the user-setpoint `.VAL` and the user-readback `.RBV` are fields of the same record:

```
motor = ophyd.EpicsSignal(
    write_pv='MOTOR.VAL',
    read_pv='MOTOR.RBV',
    name='value')
status = motor.set(3.0)
```

Signals may also be enforced to be read-only at the ophyd layer, on top of any access rights enforced at the IOC level. These signals are differentiated easily by the `RO` suffix on the class – i.e., `EpicsSignalRO`.

For example, the following is effectively `caput PV:NAME.VAL 3.0`:

* Work supported by U.S. D.O.E. Contract DE-AC02-76SF00515.
† klauer@slac.stanford.edu

[1] There is first-class `Device` support in ophyd for motor record. See `ophyd.EpicsMotor`.

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WEPHA083

```
value = ophyd.EpicsSignal(
    'PV:NAME.VAL', name='value')
status = value.set(3.0)
```

Whereas the following raises `ReadOnlyError` without reaching out to CA:

```
value = ophyd.EpicsSignalRO(
    'PV:NAME.VAL', name='value')
value.set(1.0)
```

Read-back and Setpoint Conventions

As records in EPICS largely contain a single value and its metadata, a single setpoint from the user and its associated readback value from hardware are often split into two distinct records.

Several conventions exist for easily identifying a setpoint record to a readback record solely by the name – making it trivial, even as a client with just a PV name – to determine the directionality of the record.

In areaDetector the convention for identifying a setpoint record compared to a readback record is the suffix `_RBV`. For example, the acquire time setpoint and readback PV for the simulation detector example are `13SIM1:cam1:AcquireTime` and `13SIM1:cam1:AcquireTime_RBV`, respectively.

A simple shortcut is provided in ophyd to work with these types of areaDetector PVs – `ophyd.EpicsSignalWithRBV` – which requires only the setpoint prefix. The entirety of the class is as follows, making it a thin wrapper around `EpicsSignal`:

```
class EpicsSignalWithRBV(EpicsSignal):
    def __init__(self, prefix, **kwargs):
        super().__init__(prefix + '_RBV',
                         write_pv=prefix,
                         **kwargs)
```

An example of usage might be:

```
acq = EpicsSignalWithRBV(
    '13SIM1:cam1:AcquireTime', name='acq')
```

where it's worth noting the following:

- `acquire.get()`, `acquire.read()` use the readback PV
- `acquire.put()` performs effectively a caput to the setpoint (i.e., write) PV
- `.set()` writes to the setpoint and returns a `ophyd.Status` object indicating when the readback and setpoint match (i.e., the hardware acknowledged the write and reflected the request in the readback value).

Signals in Other Control Systems

The place for other control systems which work on a signal-by-signal basis to be made compatible with ophyd is by subclassing `Signal`. For example, a Beckhoff TwinCAT3 PLC communication protocol called ADS was recently implemented for the Linac Coherent Light Source, allowing direct ophyd communication with all deployed Beckhoff PLCs.

Discussions are also underway between the ophyd developers and with facilities that primarily use Tango.

The bluesky Interface

ophyd signals support the bluesky interface directly, allowing for their usage in data acquisition routines and scanning.

Among others, `Signal` offers the following bluesky-required items [2]:

- A unique human-readable name, accessible via the `.name` attribute.
- An indication of hierarchy, noting its parent in an attribute `.parent`. Signals used in isolation have no parent.
- A `.describe()` method, indicating the type of information held in the signal, such as:

```
{
 'signal_name': {'source': 'PV:NAME',
                 'dtype': 'number',
                 'shape': [],
                 }
 }
```

- A `.read()` method, representing the data held by the signal in a consistent way:

```
{
 'signal_name': {'value': 1.3,
                 'timestamp': 0,
                 }
 }
```

  where `timestamp` would be a valid UNIX timestamp.
- A `.set()` method, which effects a "put" to the underlying control system and returns a sort of "future" – an `ophyd.Status` – object which indicates the completion status of the request.
- `.subscribe()` and `.clear_sub()` methods, allowing for monitoring of changing values with a provided callback function.

DEVICES

Devices in ophyd are intended to represent the next level above signals, in a hierarchical sense.

In creating a new device, the programmer:

- Subclasses `ophyd.Device` or a subclass thereof.
- Groups relevant signals and devices, each of which is referred to as a `Component`..
- Optionally indicates the relative importance of each component by noting a `kind`.

In practice, this means that an `ophyd.Device` can be composed of one or more `ophyd.Signal` or `ophyd.Device`.

[2] See the "Readable Device" section of `https://blueskyproject.io/bluesky/hardware.html` for more information.

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WEPHA083

Ready-to-use Devices

There are built-in device abstractions with ophyd, taking the burden of re-creating many common devices from the end-user:

- Motor record
- Scaler record
- Multi-channel analyzers and DXP from synApps
- areaDetector cameras and plugins (see Table 1)
- Simulation devices (signal, motor, detector)

Basic Form

Assume the following EPICS PVs are available from a network-accessible IOC: `Prefix:1:ItemA`, `Prefix:1:ItemB`, `Prefix:2:ItemA`, `Prefix:2:ItemB`.

A basic user-defined device might look like the following:

```
from ophyd import (Device, EpicsSignal,
                   Component)

class MyDev(Device):
    a = Component(EpicsSignal, 'ItemA')
    b = Component(EpicsSignal, 'ItemB')
```

This defines a device class named `MyDev`, which subclasses `ophyd.Device`. It has two `Components`, each of which is an `EpicsSignal`. As `Component` is of the form, `Component(class, suffix=None, **kwargs)`, the `a` component has a suffix `"ItemA"`, and the `b` component has a suffix `"ItemB"`.

The device could then be instantiated one or more times:

```
dev_1 = MyDev('Prefix:1:', name='dev_1')
dev_2 = MyDev('Prefix:2:', name='dev_2')
```

Upon instantiation, `dev_1.a` is made to be an `EpicsSignal` with the full PV `Prefix:1:ItemA`. Similarly `dev_2.b` is made to be an `EpicsSignal` with the full PV `Prefix:2:ItemB`.

The entire device could be read through the bluesky interface, e.g.:

```
dev_1.read()
```

This would read and package values and timestamps from PVs `Prefix:1:ItemA` and `Prefix:1:ItemB` into a dictionary such as:

```
{
 'dev_1_a': {'value': 0.5,
             'timestamp': 1569706862.023
             },
 'dev_1_b': {'value': 2.0,
             'timestamp': 1569706861.142
             },
 }
```

Similarly, individual components could be used as normal `EpicsSignals`:

```
dev_1.a.read()
```

Why not just EpicsSignal? In this simple case, it would be acceptable for the user to create individual `EpicsSignal` instances to communicate with the IOC without a device grouping the signals together. The convenience and utility of the abstraction shines in the scenario where one or more of the following apply:

- Many components exist (e.g., items from A–Z and not just A, B).
- Many devices exist with only a differing prefix.
- Existence of device-specific utility functions exist that should operate on one or more of the components.
- The desire to use the device in a data acquisition scenario.

Prefixes

Prefixes are additive. That is, instantiating a device with a prefix will take individual component suffixes and append them to the device prefix to make full PV names that EPICS would recognize.

This type of structuring has a few implications:

- PV names should be sensibly formed with a delimiter, with individual portions sorted from least to most specific, e.g., `FACILITY:AREA:DEVICE:COMPONENT.FIELD`.
- Device components should all share the same prefix
- Devices group signals such that they may be reused – instantiated as-is or combined and composed into a higher-level device

While the author believes that the aforementioned structure is beneficial enough to impose naming standards on the IOC level, in some facilities (or for some specific devices), it may not always be a possibility.

In such cases, it is possible to either disable the prefix-joining functionality or use a `FormattedComponent`. `FormattedComponent` allows for a format string to be used, similar to macros in IOC startup scripts or PyDM, EDM, CSS, etc screens.

Kinds, Hints, and Labels

ophyd allows for the classification of components by level of interest. This is referred to as a component `kind`. Four kinds are currently recognized: `"config"`, `"hinted"`, `"normal"`, and `"omitted"`. A single kind may be specified as a string value, or multiple kinds can be specified by bitwise or-ing `ophyd.Kind` flags.

- `"config"` – a configuration component, which is not likely to change frequently and may give an indication as to how a device was set up prior to a scan or data acquisition routine. For a detector that mostly uses fixed exposure times throughout a scan, exposure time might be considered a `"config"` component. This kind is often the majority of those seen on a device.
- `"normal"` – a component which is important enough to be read at every point during a scan. For example, a mo-

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WEPHA083

tor setpoint or readback position might be a "normal" component.
- "hinted" – a component which acts as an indicator to higher-level routines that the component could be used for plotting or in a table, etc. It also implies the "normal" flag.
- "omitted" – a component which is largely unimportant as far as data acquisition is concerned, but may be included for completeness of the device description.

Additional parts of the bluesky interface take advantage of these kinds:

- read_configuration() which reads all components marked as "config".
- describe_configuration() which describes all components marked as "config".
- hints which enumerates data fields coming from components marked as "hinted".

Versioning Devices

Devices, like all ophyd object (ophyd.OphydObj) subclasses, may carry versioning information directly in the class definition. Three parameters are currently available to describe the version:

- version – the version number itself, which should be a tuple of integers or any other unambiguously sortable value.
- version_of – the earliest versioned class.
- version_type – an indicator of what the version refers to, likely a string containing an IOC release version, detector firmware version, or similar.

For example, an IOC named XyzIOC is released with support for XyzDevice. The IOC has a 1.0 and a 2.0 release, in which the API provided by the PVs changes in some significant manner, and both are deployed at a single facility.

The device classes might look like the following:

```
class XyzDevice(ophyd.Device,
                version=(1, 0),
                version_type='XyzIOC'):
    value = Cpt(EpicsSignal, 'OldPV')
    ...

class XyzDevice_V20(XyzDevice,
                    version=(2, 0),
                    version_of=XyzDevice,
                    version_type='XyzIOC'):
    value = Cpt(EpicsSignal, 'NewPV')
    ...
```

It is then possible to programmatically select a version:

```
cls = ophyd.select_version(
    XyzDevice, (2, 1))
```

where cls is set to the latest compatible version, XyzDevice_V20.

Introspecting Devices

Devices have some built-in functionality for user convenience, allowing for inspecting devices either interactively in the Python command-line interface or programmatically. Instantiating the device from the previous section, a useful representation (referred to as a repr) is immediately available:

```
In [1]: dev = XyzDevice_V20('Dev:',
                            name='dev')

In [2]: dev
Out[2]: XyzDevice_V20(prefix='Dev:',
    name='dev', read_attrs=['value'],
    configuration_attrs=[])
```

As Device-based classes define the interface, it is not necessary to instantiate one to find information about it:

```
In [3]: XyzDevice_V20.component_names
Out[3]: ('value',)

In [4]: XyzDevice_V20.value
Out[4]: Component(EpicsSignal, 'NewPV',
                  kind='normal')
```

Components are accessible through tab completion, such that typing dev.v and pressing Tab would result in dev.value. Some additional attributes of note are:

- .parent – instantiated objects are aware of their location in the device hierarchy, including the path back to the top-level (.root) device.
- .attr_name – the dotted attribute name, allowing access to the component from the root.
- .component_names – a list of available components.
- .connected – a boolean indicator of the connectivity status of all contained components.

Composing Devices

As ophyd devices designed for reusability, devices can be composed into higher-level abstractions. Take, for example, a point detector that is atop an XY translation stage. These could be readily combined as follows:

```
from ophyd import (
    Device, Component as Cpt,
    EpicsSignalRO, EpicsMotor)

class DetAndMotor(Device):
    diode = Cpt(EpicsSignalRO, 'Diode')
    motor = Cpt(EpicsMotor, 'Motor')

both = DetAndMotor('AREA:', name='both')
```

both could then be used in such a way:

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WEPHA083


```
both.diode.read()
both.motor.velocity.set(1.0)
both.motor.velocity.kind = 'normal'
both.read()
```

This type of composition is often found to be useful by beamline end-users, as it readily allows for exploration and configuration of related devices on the command-line.

Conclusion

ophyd devices make imposing the hierarchy of devices simple, readable, and descriptive. The devices group together related signals from the underlying control system. This structure then provides a consistent interface across a wide-range of devices by design, which can then be used by higher-level software for a variety of tasks.

REFERENCES

[1] EPICS – Experimental Physics and Industrial Control System, `https://epics.anl.gov/`

[2] ophyd and the bluesky project, `https://blueskyproject.io/`

[3] areaDetector: EPICS software for area detectors, `https://cars9.uchicago.edu/software/epics/areaDetector.html`

Table 1: areaDetector Support in ophyd

Plugins	**Cameras**
AttrPlotPlugin	AdscDetectorCam
AttributePlugin	Andor3DetectorCam
CircularBuffPlugin	AndorDetectorCam
CodecPlugin	BrukerDetectorCam
ColorConvPlugin	DexelaDetectorCam
FFTPlugin	FirewireLinDetectorCam
FilePlugin	FirewireWinDetectorCam
GatherPlugin	GreatEyesDetectorCam
HDF5Plugin	Lambda750kCam
ImagePlugin	LightFieldDetectorCam
JPEGPlugin	Mar345DetectorCam
MagickPlugin	MarCCDDetectorCam
NetCDFPlugin	PSLDetectorCam
NexusPlugin	PcoDetectorCam
Overlay	PerkinElmerDetectorCam
OverlayPlugin	PilatusDetectorCam
PluginBase	PixiradDetectorCam
PosPlugin	PointGreyDetectorCam
ProcessPlugin	ProsilicaDetectorCam
PvaPlugin	PvcamDetectorCam
ROIPlugin	RoperDetectorCam
ROIStatPlugin	SimDetectorCam
ScatterPlugin	URLDetectorCam
StatsPlugin	
TIFFPlugin	
TimeSeriesPlugin	
TransformPlugin	