# New Module Control Interfaces for Python

**Philipp Wieduwilt**[1], Harrison Schreeck[1]

[1]Universität Göttingen

*philipp.wieduwilt@phys.uni-goettingen.de*

October 8, 2018


# Motivation

- The interface for module control between (calibration) software and hardware is realized via EPICS.
- We use pyepics for reading and writing EPICS **P**rocess **V**ariables.
- Still the calibration software (scripts) are hard to understand and maintain:
  - PV names are often not very verbose
  - some actions (e.g. activate ACMC) require setting multiple PVs
  - some actions require certain sequences of PV writes

$\Rightarrow$ Add a layer of abstraction while maintaining possibility for low level PV actions.

# New Python classes

## lib/asics.py

- configuration of ASICs: DCD, DHP, Switcher classes
- should normally not be used directly but rather via **device** instance
- writes PVs directly

## lib/devices.py

- configuration of modules: ASICs and DHE (DHI?)
- has X instances of **ASIC** attached
- writes PVs via a queue, dispatch needed

## lib/dhh.py

- DHC class: modules and DHC configuration
- has X instances of **device** attached

# The Device class

```
from devices import Device
module = Device("H1011", device_type="pxd9",
                        module_flavor="auto",
                        module_name="auto",
                        ps="auto",
                        dcdversion=42, dhpversion=12)
)
```

- supported device types are 'pxd9' and 'hybrid5'
- favor ('if', 'ib', ...) will be deduced from DHE name if 'auto'
- module name will be same as DHE name if 'auto'
- PS name will be same as DHE prefixed with 'P' if 'auto'

# Device subclasses

- There are friendlier subclasses of **Device** that can be used:

```
import devices
myHybrid5 = devices.Hybrid5("H1011",
                    module_name="auto",
                    ps="auto",
                    dcdversion=42, dhpversion=12)
myPXD9 = devices.PXD9("H1011",
                    module_flavor="auto",
                    module_name="auto",
                    ps="auto",
                    dcdversion=42, dhpversion=12)
)
```

# Usecase: passing of device information

- Use **Device** instance for passing information to functions.

```
def some_function(module):
        dhe_name = module.dhe
        ps_name = module.ps
        device_type = module.device_type
        # ASIC information
        number_of_dcds = len(module.dcds)
        dcd1_version = module.dcd1.version
        for dhp in module.dhps:
                print dhp.version
)
```

# Usecase: high-level configuration

- Use high-level functions for configuring the module.

```
# voltages and currents
module.set_voltage("gate-on1", -2000)
module.set_current("clear-off", 20)
# high speed links
if not all(module.get_links()):
        print "alarm"
# check link status, reset if necessary, try 3 times
module.check_links(short_rst=True, trials=3)
# analog common mode correction
print module.get_acmc() # -> ['on','on','off','off']
module.set_acmc(1, dcd=3)  # enable acmc on DCD3
# DCD JTAG out
module.set_dcd_jtag(0)
```

# Usecase: mid-level configuration

- Use mid-level functions for configuring the module.

```
# set ASIC PVs
module.set_dcd_pvs([("testinj:S:set", 1),
                    ("en30:S:set", 0),
                    ("en90:S:set", 1)],
                   dcd=0)
module.set_dcd_register("global")
module.set_dhp_register("core", dhp=1)
# dcd=0 or dhp=0 is default and sets all
```

# Usecase: low-level configuration

- Use low-level functions for configuring ASICs.
- Need to **dispatch** for writing PVs to the system!

```
for dcd in module.dcds:
        dcd.set_pv("dacipsource:VALUE:set", 80)
        dcd.set_register("global")
module.dispatch() # writes PV queue

module.dhp4.set_switch("active_ped_mem", 1)
module.dhp4.set_register("core")
module.dispatch()
```

# Dispatch and PV queue

- **Device** instance uses internal queue[1] for handling PV writes.
- PVs are only written once `dispatch()` method is called.
- High-level class methods include a dispatch, only low-level methods (e.g. `set_pv()`) need manual dispatch and register writes.

There are possible improvements to this:

## multi-queue

- use one queue per register per ASIC
- `dispatch()` writes all non-empty registers
- overall **order** of PV writes is no longer chronological

[1] not DHE JTAG queue!

# Dispatch and PV queue (2)

## connect all PVs on startup

- reason for not writing PVs directly: slow PV connection

$\rightarrow$ connect all PVs on `init()`

- possible?

## PV manager

- maintain database of PV $\leftrightarrow$ register
- `dispatch()` triggers PV manager to sort PV queue, write PVs and trigger register writes as necessary
- will introduce an **overhead** during PV writing

# DHC support

- `dhh.DHC` class adds DHC support.
- Mostly dummy functions at the moment.

```
import devices
import dhh

myDHC = dhh.DHC("H10")
mod = devices.PXD9("H1011")

myDHC.add_module(mod)
myDHC.set_dhe_mask("0001")
```

# Next steps

- Testing and bug hunting.
- Port all `*_utils.py` functions.
- Make classes thread-safe for threadding/multi-processing usage.
- Integrate DAQ into Device/DHC class?
- Port calibration scripts $\rightarrow$ maintainability
- Add e.g. calibration class??

# Backup