Particle Accelerator MIddle LAyer (PAMILA)

Yoshiteru Hidaka

NSLS-II, BNL

November 15, 2024 @ python Accelerator Middle Layer Virtual Meeting

Brookhaven National Laboratory

U.S. DEPARTMENT OF ENERGY

@BrookhavenLab

Motivation / Main Ideas for New Middle Layer

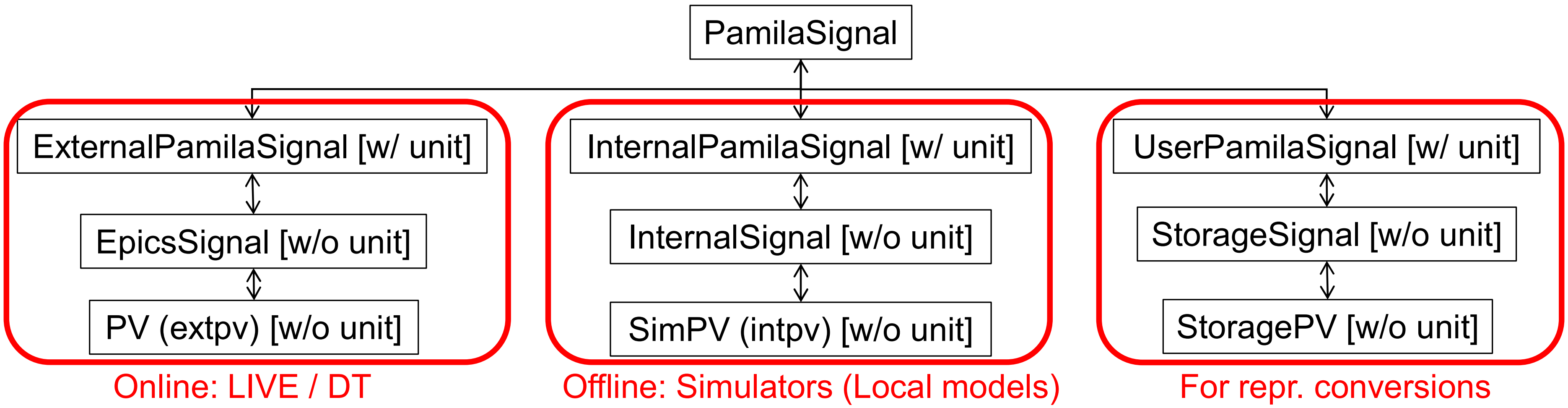
- Initially setting up for `pyacal`, but realized simulation mode not implemented yet, and only work with a digital twin (DT) at that time.
- Started creating a minimal working DT and also implementing a simulator mode with `pyAT`. ➔ Turned out not a quick job.
- At some point, I thought I might as well write a concept code that satisfies my wish list for a new middle layer...:
  - Can handle any type of facility-specific customization for unit conversion (including multi-input-multi-output).
  - Compatible with `bluesky` / `ophyd` / `tiled`
    - Push facility-specific customizations to low levels to make customizations invisible in the logic of high-level code as much as possible.
    - Utilize modern data management.
  - More modular high-level applications (HLAs)
    - More reusable / manageable parameter specifications.

Unit Conversion

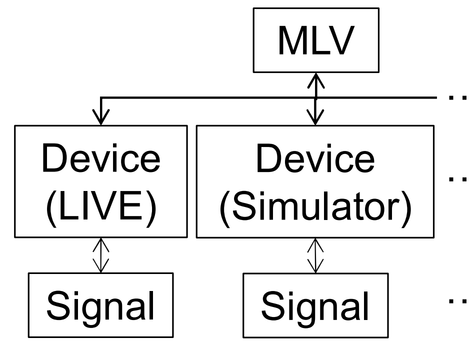
- PAMILA distinguishes 2 types of unit conversion:
  - Universal unit conversion
    - NOT device dependent: “intra-dimensional” unit conv. (i.e., unit dimension DOESN’T change)
    - Examples: mA ⇔ A, mm ⇔ nm, GHz ⇔ Hz
    - Handled at PAMILA signal level via Python package “`pint`”
  - Representation conversion
    - Device dependent
      - Typically, “inter-dimensional” unit conv. (i.e., unit dimension DOES change)
    - Examples: A ⇔ mrad, A ⇔ m^{-1}, digital counts ⇔ A, etc.
    - Handled at PAMILA device level
      - Use “repr” (representation) to avoid confusion
        - “unit”: [A], [mA], [m^{-1}], etc.; no association to devices
      - Example: Combined func. magnet w/ Ch.1 and Ch.2 currents [A] that affect b1 [rad] and b2 [m^{-1}] has repr’s :
        - I_1 [A], I_2 [A], b_1 [rad], b_2 [m^{-1}]

Brookhaven National Laboratory

PAMILA Class Hierarchy

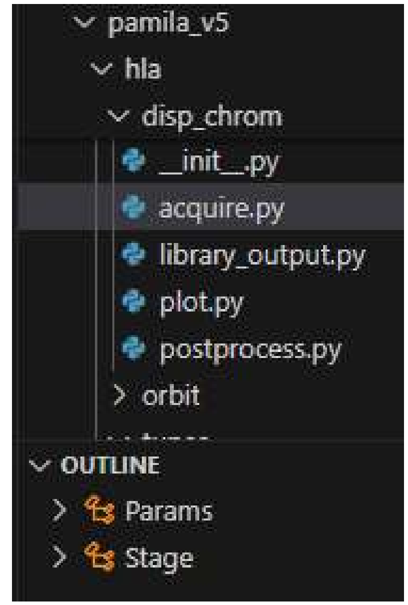


- `PamilaDevice (pdev)`
  - Contains a custom `ophyd Device` object as an attribute (i.e., `bluesky` compatible)
  - The `ophyd` device takes `PamilaSignal`'s as components
  - Handles repr. conversions
- `MiddleLayerVariable (mlv)`
  - Holds one `pdev` for each <u>machine mode</u> (LIVE, DigitalTwin, SIMULATOR, …)
- Asynchronous (parallel) get/put operations for convenience:
  - `MiddleLayerVariableList (mlvl)`: A list of `mlv`'s with enable/disable control
  - `MiddleLayerVariableTree (mlvt)`: Each attribute points to `mlvl`'s
- End users are expected to deal with only `mlv` and above, NOT `PamilaDevice` / `PamilaSignal` objects.



High-Level Application (HLA) into Stages

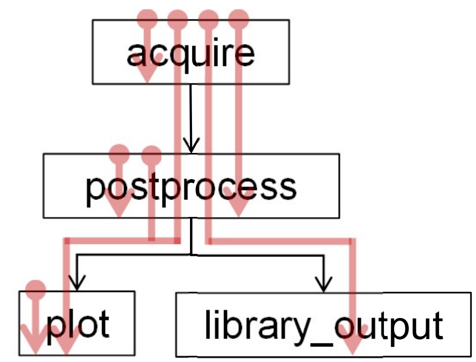
- Typical stages for an HLA
  - acquire
    - Input: Data acquisition options
    - Output: Raw data (w/ or w/o tiled uid)
  - postprocess
    - Input: Raw data
    - Output: Postprocessed data (w/ or w/o tiled uid)
  - plot
    - Input: Raw/postprocessed data
    - Output: Plots
  - library_output
    - Input: Raw/postprocessed data
    - Output: Data in a format consumable by other HLAs



- Each stage implemented as a module. Within each module:
  - A "Params" object specifies all options for a "Stage" object.
  - To run a stage, just call "run" method of the "Stage" object.

HLA Flow

- An HLA flow (sequencer) specifies a sequence of HLA stages to be run.
- Calling "run" method of this "flow" object starts running the first stage, passes the output to the next stage, runs the next stage, and repeats until the final stage.
- A "flow" can enter and exit at any stage
  - Easy re-processing of raw data with different post-processing options.
  - Enable stage-by-stage debugging.
  - Easy switch between standalone and library usage.
- Another benefit: Hierarchical parameter (option) specifications
  - avoid cluttering with many "flat" (and often irrelevant) options



Example: Dispersion/Chormaticity HLA

HLA "disp_chrom" - Stage "acquire":

```
class Params(HlaStageParams):
    rf_freq_mlv_SP: MiddleLayerVariable | MlvName | str = Field(MACHINE_DEFAULT)
    rf_freq_mlv_RB: MiddleLayerVariableRO | MlvName | str | None = Field(None)
    orbit_meas: HlaFlow | None = Field(MACHINE_DEFAULT)
    tune_meas: HlaFlow | None = Field(MACHINE_DEFAULT)
    n_freq_pts: int = Field(5, ge=2)
    max_delta_freq: Q_ = Field(Q_("200 Hz"))
    min_delta_freq: Q_ = Field(Q_("-200 Hz"))
    extra_settle_time: Q_ = Field(Q_("0 s"), ge=Q_("0 s"))
    use_bluesky: bool = Field(False)
    save_to_tiled: bool = Field(False)
```

HLA "orbit/slow_acq" - Flow "library" - Stage "acquire":

```
class Params(RepeatMeasHlaStageParams):
    n_meas: int = Field(5, ge=1, description="Number of orbit measurements to acquire")
    wait_btw_meas: Q_ = Field(Q_("0.2 s"), ge=Q_("0 s"),
                              description="Wait time between each measurement")
    stats: StatisticsType | Sequence[StatisticsType] = Field((StatisticsType.MEAN,
                                                              StatisticsType.STD,
                                                              StatisticsType.MIN,
                                                              StatisticsType.MAX))
    bpm_mlo: MiddleLayerVariableListRO | MiddleLayerVariableTree | MloName | str = Field(
        MACHINE_DEFAULT)
    use_bluesky: bool = Field(False)
    save_to_tiled: bool = Field(False)
```

HLA "disp_chrom" - Stage "postprocess":

```
class Params(HlaStageParams):
    momentum_compaction: float | DesignLatticeProperty = Field(MACHINE_DEFAULT)
    disp_max_order: int = Field(2, ge=1, description="Max order for dispersion fitting")
    chrom_max_order: int = Field(2, ge=1, description="Max order for chromaticity fitting")
```

HLA "tunes/via_pvs" - Flow "library" - Stage "acquire":

```
class Params(RepeatMeasHlaStageParams):
    n_meas: int = Field(3, ge=1, description="Number of measurements to acquire")
    wait_btw_meas: Q_ = Field(Q_("1.0 s"), ge=Q_("0 s"),
                              description="Wait time between each measurement")
    stats: StatisticsType | Sequence[StatisticsType] = Field((StatisticsType.MEAN,
                                                              StatisticsType.STD,
                                                              StatisticsType.MIN,
                                                              StatisticsType.MAX))
    tune_mlvt: MiddleLayerVariableTree | MlvtName | str = Field(MACHINE_DEFAULT)
    use_bluesky: bool = Field(False)
    save_to_tiled: bool = Field(False)
```

HLA "disp_chrom" - Stage "plot":

```
class Params(HlaStageParams):
    show_plot: bool = Field(True)
    title: str = Field("")
    export_to_file: Path | None = Field(None)
```

Main script:

```
SR = pml.load_machine("SR", dirpath=facility_folder)
pml.load_hla_defaults(facility_folder / "hla_defaults.yaml")
pml.set_online_mode(MachineMode.DIGITAL_TWIN)
pml.go_online()
standalone = pml.hla.disp_chrom.get_flow("standalone", SR)
params = standalone.get_params("plot")
params.export_to_file = Path("test.pdf")
standalone.run()
```

disp_chrom/__init__:

```
match flow_type:
    case "library":
        stage_classes = [acquire.Stage, postprocess.Stage, library_out
    case "standalone":
        stage_classes = [acquire.Stage, postprocess.Stage, plot.Stage]
```

Brookhaven National Laboratory

bluesky Wrapper

- A 30-min. meeting on setting up `tiled` with D. Allan & M. Rakitin from NSLS-II DSSI
- Need some getting used to `yield/yield from` in `bluesky` plans
- Developed wrapper functions:
  - `get(obj_list_to_get)`
  - `abs/rel_put(obj_list_to_put, values_to_put)`
  - `abs/rel_put_then_get(obj_list_to_get, obj_list_to_put, values_to_put)`
- `bluesky`'s RunEngine runs plans within the wrapper functions
- Working with monkey patching (to handle `pint` objects, etc.)
- Can write to `tiled`; can also directly get output data in memory
- Objects can be `ophyd` device/signal, `pamila` device, MLV, MLVL, and MLVT.
- Built-in capabilities of repeated measurements and statistical calculations
- Will "set and wait" (wait conditions specified in each device)
- Set modes: jump (i.e., one-step) or ramp (i.e., multi-step)
- Need integration into HLAs

Summary

- A new middle layer package "PAMILA" is being developed at NSLS-II.
- New main features include:
  - Handle complicated multi-input-multi-output "unit conversions"
  - "Flow of stages" concept for HLA implementations
  - `bluesky` / `ophyd` / `tiled` compatibility
- Initial commit for the package will be uploaded to GitHub after (a lot of) cleanup.

Backup

Desired Basic Interactions: One-to-One

mlv := Middle Layer Variable (= an abstract version of PV)
Q := Quantity object in Python `pint` unit handling package

- One-to-one `get`:
  - mlv_orbcor_x_I_RB.get() ➔ Q("0.5 A")
  - mlv_orbcor_x_angle_RB.get() ➔ Q("-10 urad")
- One-to-one `put`:
  - mlv_orbcor_x_I_SP.put(Q("0.5 A"))
  - mlv_orbcor_x_angle_SP.put(Q("-10 urad"))

- Differences from `aphla`/`pytac`:
  - Units are attached.
  - No need to specify which property, SP/RB, & unit system on every get/put()

Desired Basic Interactions: Many-to-Many (1/2)

- x-y coupled orbit corrector:
  - (Ch.1 [A], Ch. 2 [A]) determines x [urad] & y [urad] kick angles.
    - mlv_ch1_ch2.get() ➔ [Q("0.1 A"), Q("-0.2 A")] (2 PVs ➔ 2 out)
    - mlv_ch1_ch2.put([Q("0.1 A"), Q("-0.2 A")]) (2 in ➔ 2 PVs)
    - mlv_x_y.get() ➔ [Q("5 urad"), Q("-7 urad")] (2 PVs ➔ 2 out)
    - mlv_x_y.put([Q("5 urad"), Q("-7 urad")]) (2 in ➔ 2 PVs)
    - mlv_x.get() ➔ Q("5 urad") (2 PVs ➔ 1 out)
    - mlv_x.put([Q("5 urad")]) (1 in [+ 2 aux. (= Ch.1 & Ch.2)] ➔ 2 PVs)
      - This MLV defined to maintain current "y" angle
      - Enable orthogonal kick response measurements
      - Can handle ID correctors whose kick strengths are also gap dependent: 1 in [+3 aux. (= Ch.1, Ch.2 & gap)] ➔ 2 PVs
- Coupled combined function magnets (i.e., bend + quad) can be handled similarly

Desired Basic Interactions: Many-to-Many (2/2)

- 30 sextupoles in SL1 family at NSLS-II are powered by 5 independent power supplies (PS)
- Read / set the entire family of magnets:
  - mlv_SL1_I.get() ➔ [31.73[A], 31.75[A], 31.76[A], 31.76[A], 31.79[A]]
  - mlv_SL1_I.put([31.73[A], 31.75[A], 31.76[A], 31.76[A], 31.79[A]])
  - mlv_SL1_K2.get() ➔ [-13.30[m^{-3}], -13.31[m^{-3}], -13.29[m^{-3}], -13.30[m^{-3}], -13.31[m^{-3}], … (a total of 30 values)]
  - mlv_SL1_K2.put([30 values]) ➔ Not feasible / allowed
  - mlv_SL1_avg_K2.get() ➔ Q("-13.3 m^{-3}")
  - mlv_SL1_avg_K2.put(Q("-13.3 m^{-3}"))
- Six NSLS-II SL1 sextupoles are powered in series by one PS
  - mlv_SL1_G1_I.get() ➔ 31.73 [A]
  - mlv_SL1_G1_I.put(31.73 [A])
  - mlv_SL1_G1_K2.get() ➔ [-13.30[m^{-3}], -13.31[m^{-3}], -13.29[m^{-3}], -13.30[m^{-3}], -13.31[m^{-3}], 13.32[m^{-3}]]
  - mlv_SL1_G1_K2.put([6 values]) ➔ Not feasible / allowed
  - mlv_SL1_G1_avg_K2.get() ➔ Q("-13.3 m^{-3}")
  - mlv_SL1_G1_avg_K2.put(Q("-13.3 m^{-3}"))

Single-Input Single-Output (SISO) get/put

float [unit] := Python float (i.e., no unit attached) in the unit of “unit”

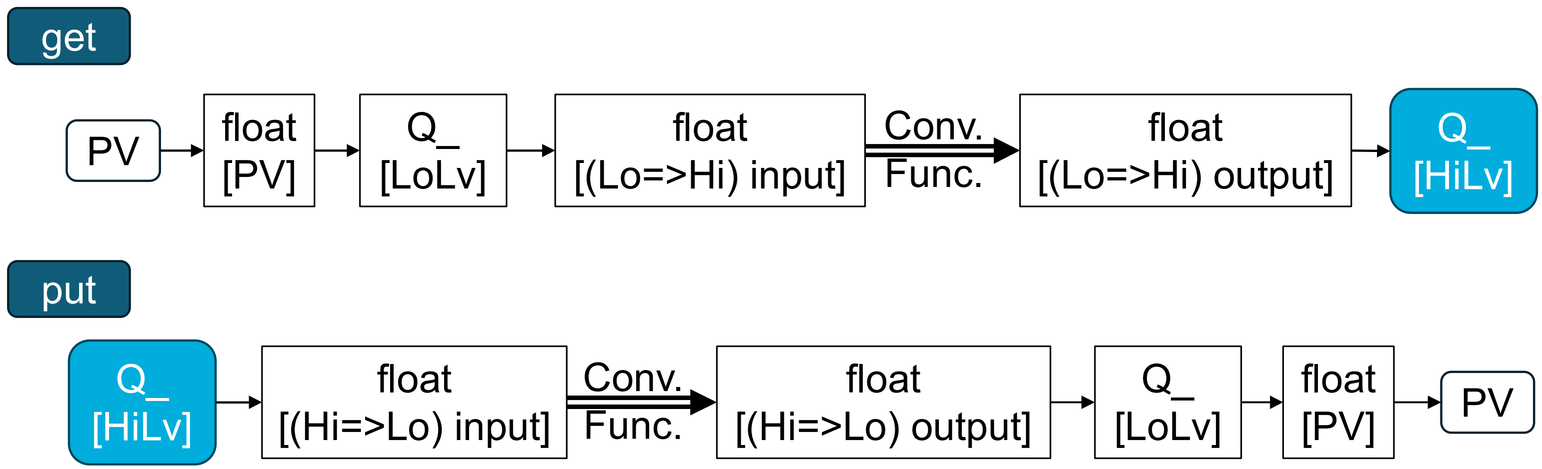
Q_ [repr] := `pint` Quantity object in the unit of “repr”

“LoLv” repr := repr at low (PV) level

“HiLv” repr := repr at high (user interaction) level

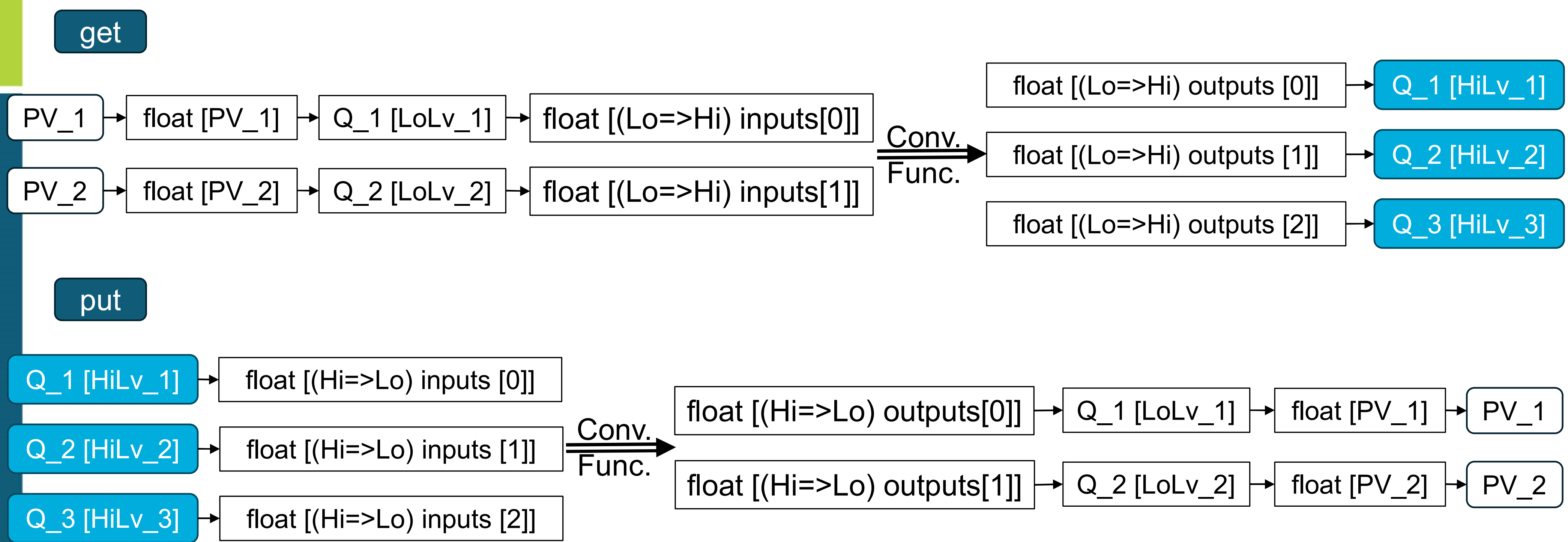
“(Lo => Hi)” := Conversion function from LoLv to HiLv

“(Hi => Lo)” := Conversion function from HiLv to LoLv



- “put” is just the reverse flow of “get”, but with the inverse of the “get” conversion function, if it exists.
- If the inverse of the “get” unit conversion function is not a function, you can still define a function with some restrictions. Or make “put” unavailable (i.e., read-only).

Multiple-Input Multiple-Output (MIMO) get/put



- “put” is just the reverse flow of “get”, but with the inverse of the “get” conversion function, if it exists.
- If the inverse of the “get” unit conversion function is not a function, you can still define a function with some restrictions. Or make “put” unavailable (i.e., read-only).

List of Re-usable HLAs

- assert_beam_current (TODO)
  - min_current, max_current
- orbit/slow_acq, fast_acq (TODO), tbt_acq (TODO)
- disp_chrom
- tunes/via_pvs & /via_tbt (TODO)
- respmat (TODO) / respmat_orb (TODO) / respmat_tune (TODO), etc.
- …