Hardware Triggered Scanning: Scanning Control

Philip Taylor, Emma Arandjelovic

Observatory Sciences Limited

Overview

- This topic will cover how to co-ordinate multiple device blocks to perform a scan

  https://pymalcolm.readthedocs.io/en/latest/tutorials/scanning.html

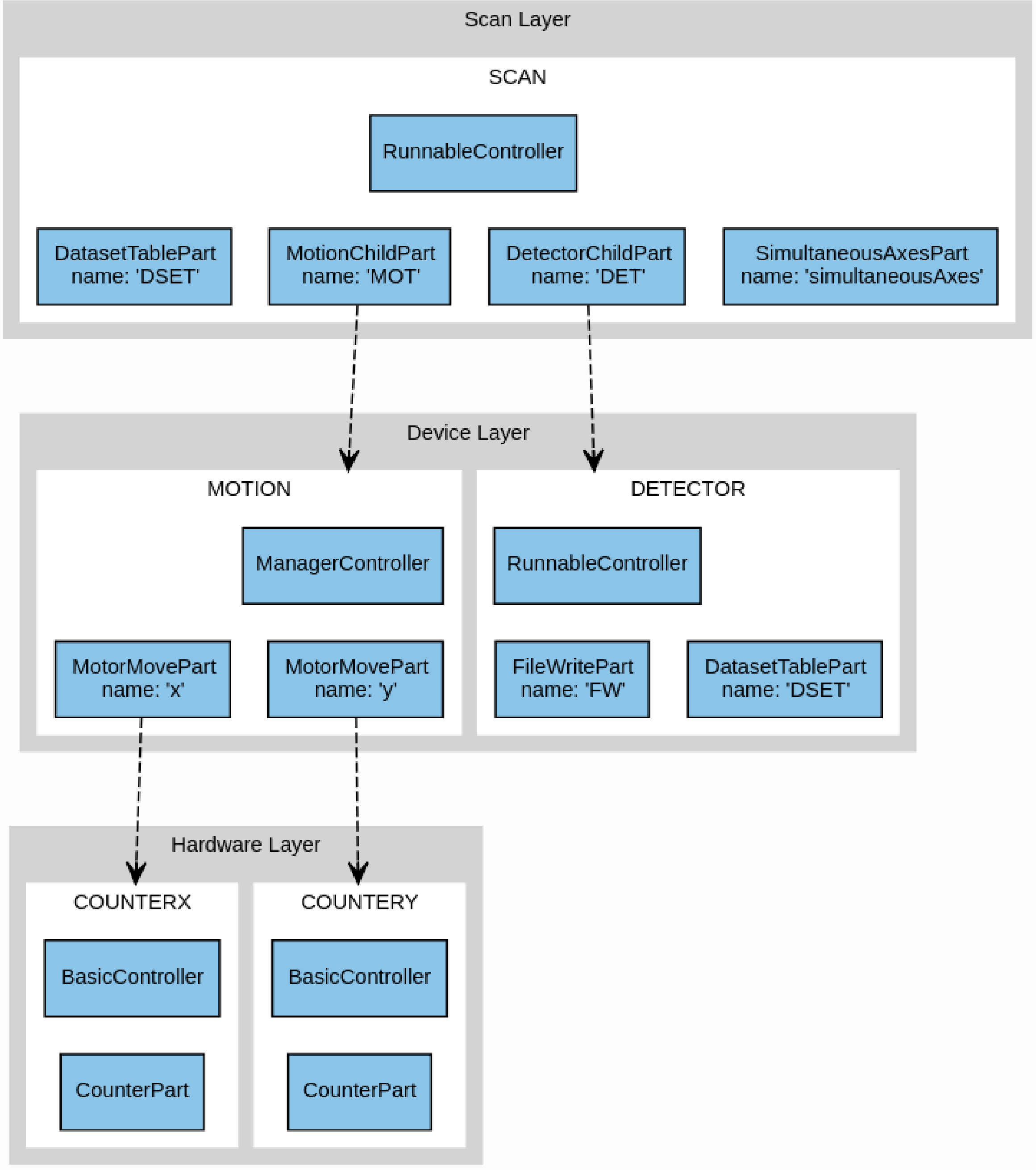
- We will also look at Malcolm's role in configuring detectors and AreaDetector plugin chains

  https://pymalcolm.readthedocs.io/en/latest/tutorials/areadetector.html

Scanning Demo

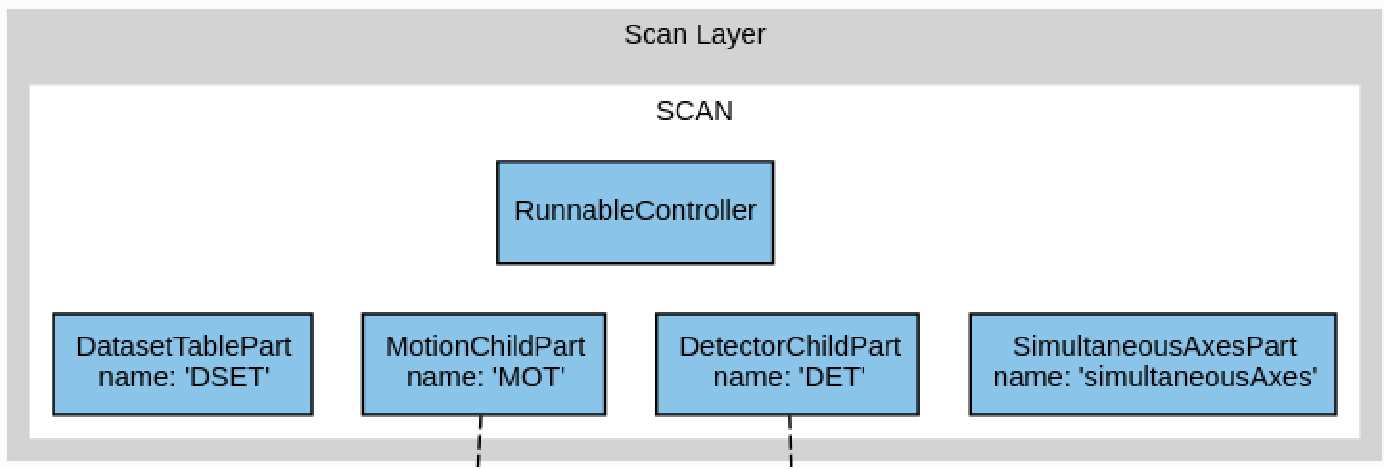
- So far we have created *hardware blocks* and *device blocks*
- The next step is to co-ordinate device blocks by creating an additional *scan layer block*
- Like the detector, this uses a *RunnableController*:
  - configure(params)
    - Configure all children, report child datasets
  - run()
    - Start children running simultaneously, monitor status and report progress

Block Hierarchy



SCAN Block Functionality

- The SCAN block uses two parts to control its children (MOT and DET)
- A new *SimultaneousAxesPart* verifies the requested axes can be scanned simultaneously
- A *DatasetTablePart* reports the child datasets


Scan Layer
SCAN
RunnableController
DatasetTablePart name: 'DSET'
MotionChildPart name: 'MOT'
DetectorChildPart name: 'DET'
SimultaneousAxesPart name: 'simultaneousAxes'

Process Definition

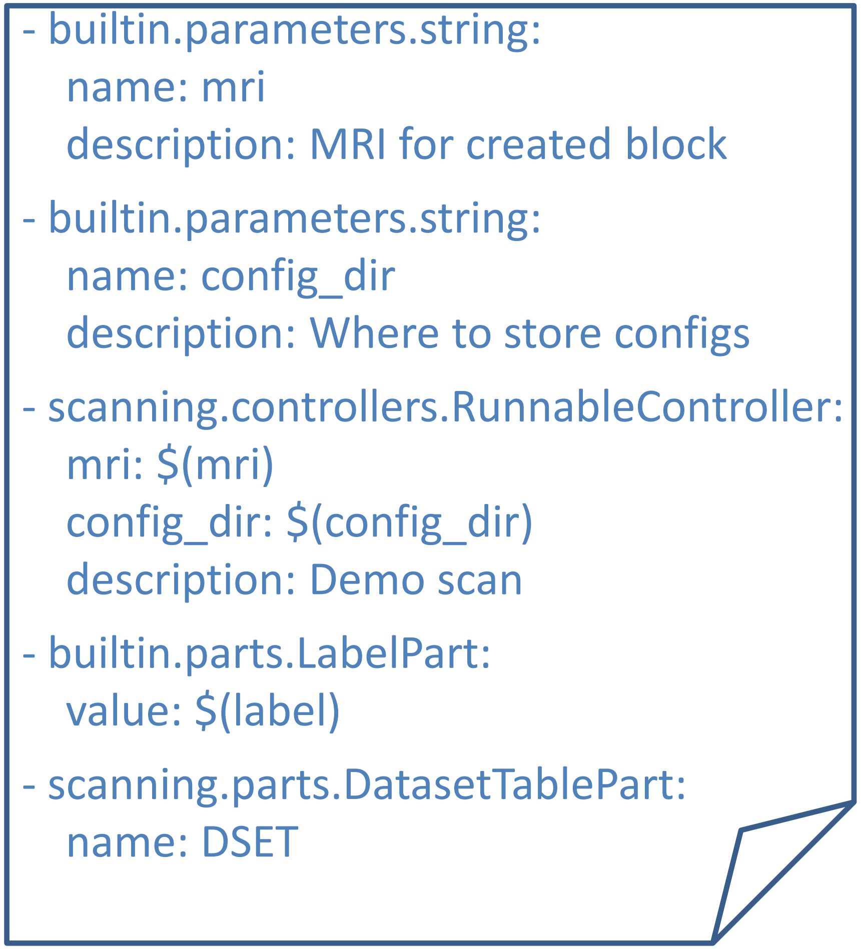
demo/DEMO-SCANNING.yaml (extract)

```
- builtin.defines.string:
    name: config_dir
    value: /tmp

- demo.blocks.motion_block:
    mri: MOTION
    config_dir: $(config_dir)

- demo.blocks.detector_block:
    mri: DETECTOR
    config_dir: $(config_dir)

- demo.blocks.scan_1det_block:
    mri: SCAN
    config_dir: $(config_dir)
    label: Mapping x, y with demo det.
```

- Create a config_dir variable that can be shared between blocks
- Instantiate the motion and detector device blocks as before
- Instantiate a new scan block to sit on top

Scan Block Definition

demo/blocks/scan_1det_block.yaml (Extract)

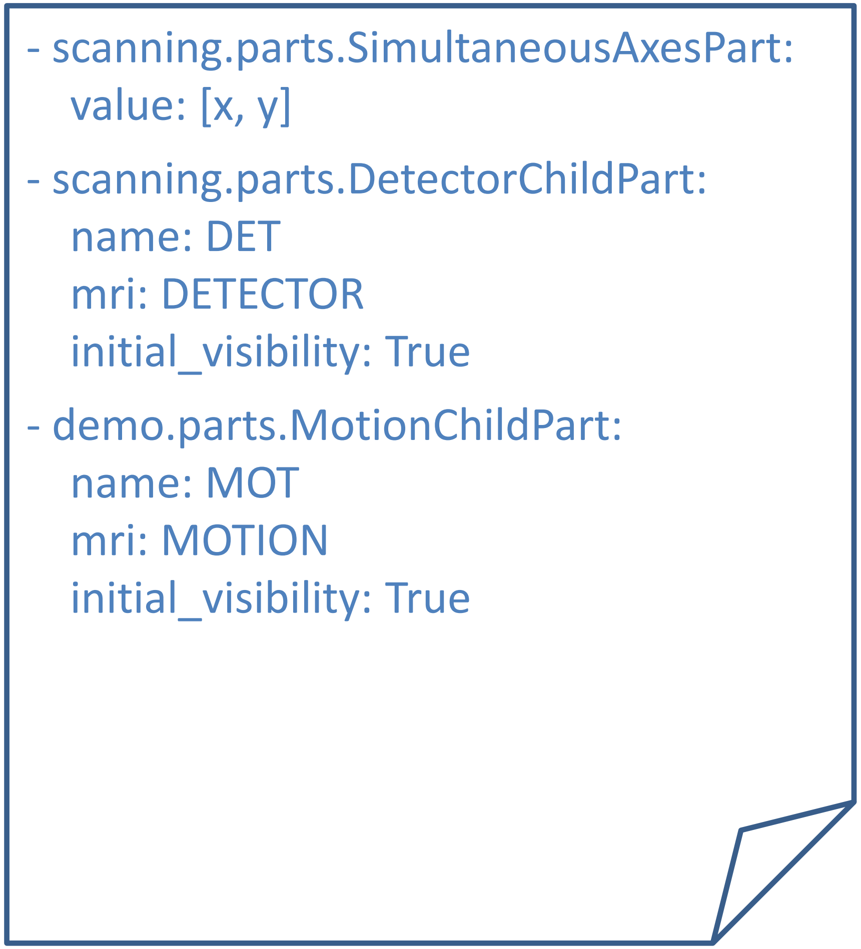


```yaml
- builtin.parameters.string:
    name: mri
    description: MRI for created block

- builtin.parameters.string:
    name: config_dir
    description: Where to store configs

- scanning.controllers.RunnableController:
    mri: $(mri)
    config_dir: $(config_dir)
    description: Demo scan

- builtin.parts.LabelPart:
    value: $(label)

- scanning.parts.DatasetTablePart:
    name: DSET
```

- Specify the parameters to be supplied
- Use a *RunnableController* to construct the block
- Instantiate a *LabelPart and DatasetTablePart* as before

Scan Block Continued

demo/blocks/scan_1det_block.yaml



```
- scanning.parts.SimultaneousAxesPart:
    value: [x, y]
- scanning.parts.DetectorChildPart:
    name: DET
    mri: DETECTOR
    initial_visibility: True
- demo.parts.MotionChildPart:
    name: MOT
    mri: MOTION
    initial_visibility: True
```

- ➢ Instantiate a new part which checks the requested motion axes are allowed to be scanned simultaneously
- ➢ Create parts for controlling the child blocks, passing in the MRI for each one

New Hook

- As before, we are going to hook into ConfigureHook, RunHook, ResumeHook etc.

- New hook: **PreConfigureHook**
  - Called at the start of configure()
  - Use the superclass ChildPart reload() function to load the last saved design to the child block

New configure() Parameter

- We are also going to add a new parameter to the configure() method:

  exceptionStep #type: int

  - Raise an exception if the scan gets to this step

- How to set this up?
  - Recall the PartRegistrar object used to register methods and attributes with the parent controller
  - Call registrar.report(info) to inform it about additional configure parameters

Motion Child Part: setup()

demo/parts/motionchildpart.py (extract):

```
class MotionChildPart(builtin.parts.ChildPart):
def setup(self, registrar):
    # type: (PartRegistrar) -> None
    super(MotionChildPart, self).setup(registrar)
    registrar.hook(scanning.hooks.PreConfigureHook, self.reload)
    registrar.hook((scanning.hooks.ConfigureHook,
            scanning.hooks.PostRunArmedHook,
            scanning.hooks.SeekHook), self.configure)
    registrar.hook((scanning.hooks.RunHook,
            scanning.hooks.ResumeHook), self.on_run)
    # Tell the controller to expose some extra configure parameters
    registrar.report(scanning.hooks.ConfigureHook
            .create_info(self.on_configure))
```

Motion Child Part: on_configure(…)

demo/parts/motionchildpart.py (continued):

```
@add_call_types
 def on_configure(self,
          completed_steps,      # type: scanning.hooks.ACompletedSteps
          steps_to_do,          # type: scanning.hooks.AStepsToDo
          generator,            # type: scanning.hooks.AGenerator
          axesToMove,           # type: scanning.hooks.AAxesToMove
          exceptionStep=0,      # type: AExceptionStep
          ):
     # Store the parameters inside the class
     self._generator = generator
     self._completed_steps = completed_steps
     self._steps_to_do = steps_to_do
     self._exception_step = exceptionStep
     self._axes_to_move = axesToMove
     self._movers = {axis: MaybeMover(child, axis) for axis in axesToMove}
```

Motion Child Part: on_run()

- ➢ How to be notified when both motors have finished moving?

- *<method>_async* is an asynchronous method which kicks off the specified Method and returns a *Future* object that can be waited on

- These can be used to start a number of long running processes simultaneously

- The MaybeMover helper class defines an asynchronous method, *maybe_move_async()*, which will move the motor if the demand position differs from the current position

Motion Child Part: on_run()

demo/parts/motionchildpart.py:

```
def on_run(self, context):
  # type: (scanning.hooks.AContext) -> None
  # Start time so everything is relative
  for i in range(self._completed_steps, self._completed_steps + self._steps_to_do):
    …
    fs = []
    for axis, mover in self._movers.items():
        mover.maybe_move_async(fs, point.lower[axis])
        mover.maybe_move_async(fs, point.upper[axis], move_duration)
    # Wait for the moves to complete
    context.wait_all_futures(fs)
    # Update the point as being complete
    self.registrar.report(scanning.infos.RunProgressInfo(i + 1))
    # If this is the exception step then blow up
    assert i + 1 != self._exception_step, \
        "Raising exception at step %s" % self._exception_step
```

Preparing the Example

```
./malcolm/imalcolm.py
malcolm/modules/demo/DEMO-SCANNING.yaml
```

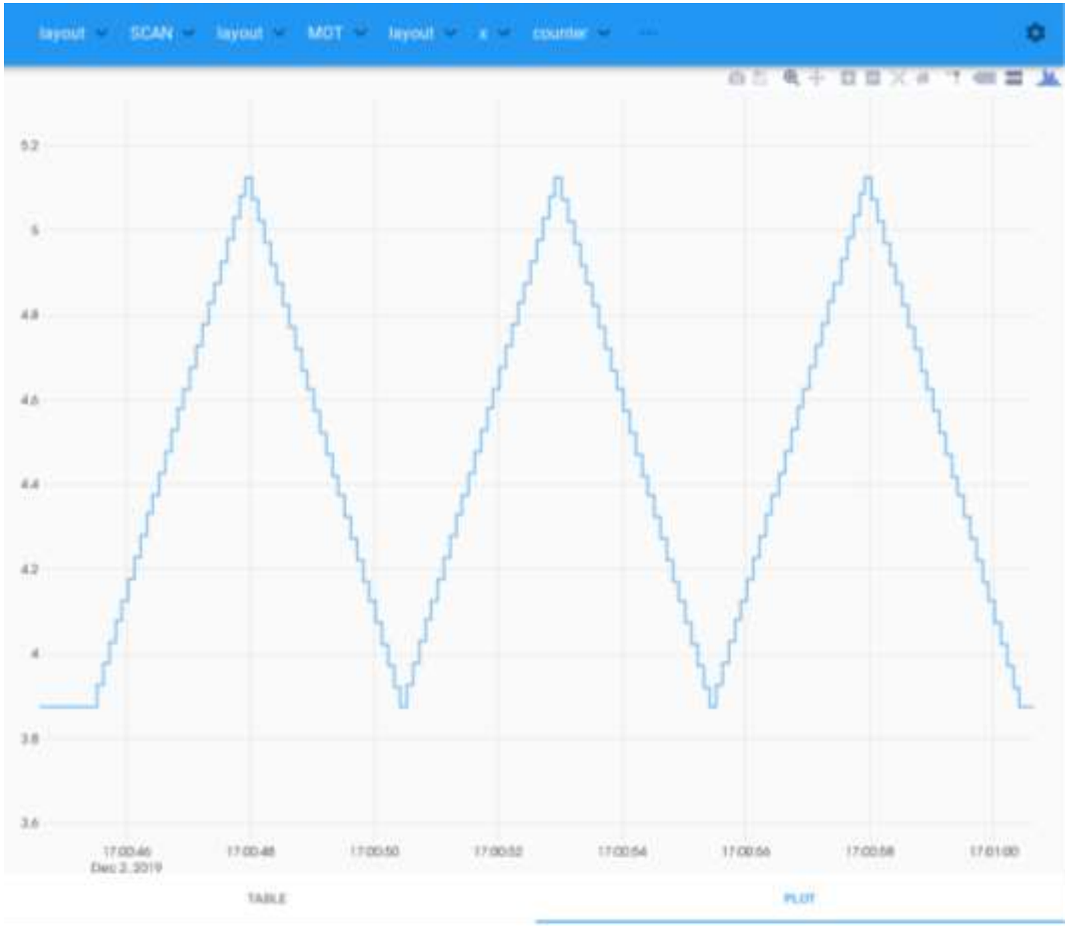
```
>>> from scanpointgenerator import LineGenerator, CompoundGenerator
>>> from annotypes import json_encode
>>> yline = LineGenerator("y", "mm", -1, -0, 6)
>>> xline = LineGenerator("x", "mm", 4, 5, 5, alternate=True)
>>> generator = CompoundGenerator([yline, xline], duration=0.5)
>>> json_encode(generator)
```

- Copy the JSON output to the clipboard

Running the Example

- Open http://localhost:8008/gui/SCAN
- Expand the Configure method and edit the generator field to paste in the JSON code
- Set fileDir to /tmp
- Press Configure and then Run
- Watch the MOTION counter blocks as they perform the snake scan
- Check a new HDF file is written to /tmp/DET.h5 by the DETECTOR device block

Scanning Demo Screenshot



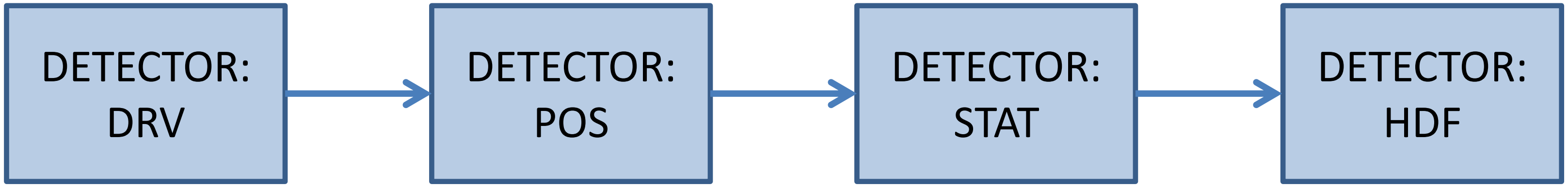
Controlling Detectors

- EPICS AreaDetector is responsible for setting up the detector and writing the data
- Malcolm’s role is supervisory:
  - Configures the plugin chain
  - Sets up the detector parameters
  - Starts acquiring
- Each detector is a ‘runnable’ device block
- The detector driver and plugins are hardware blocks

AreaDetector Demo

- We will use the AreaDetector *simDetector*
- Use case: multi-dimensional continuous scan
  - – Read the data from the simulated detector
  - – Calculate statistics
  - – Write them to a HDF5 file

Plugin Chain



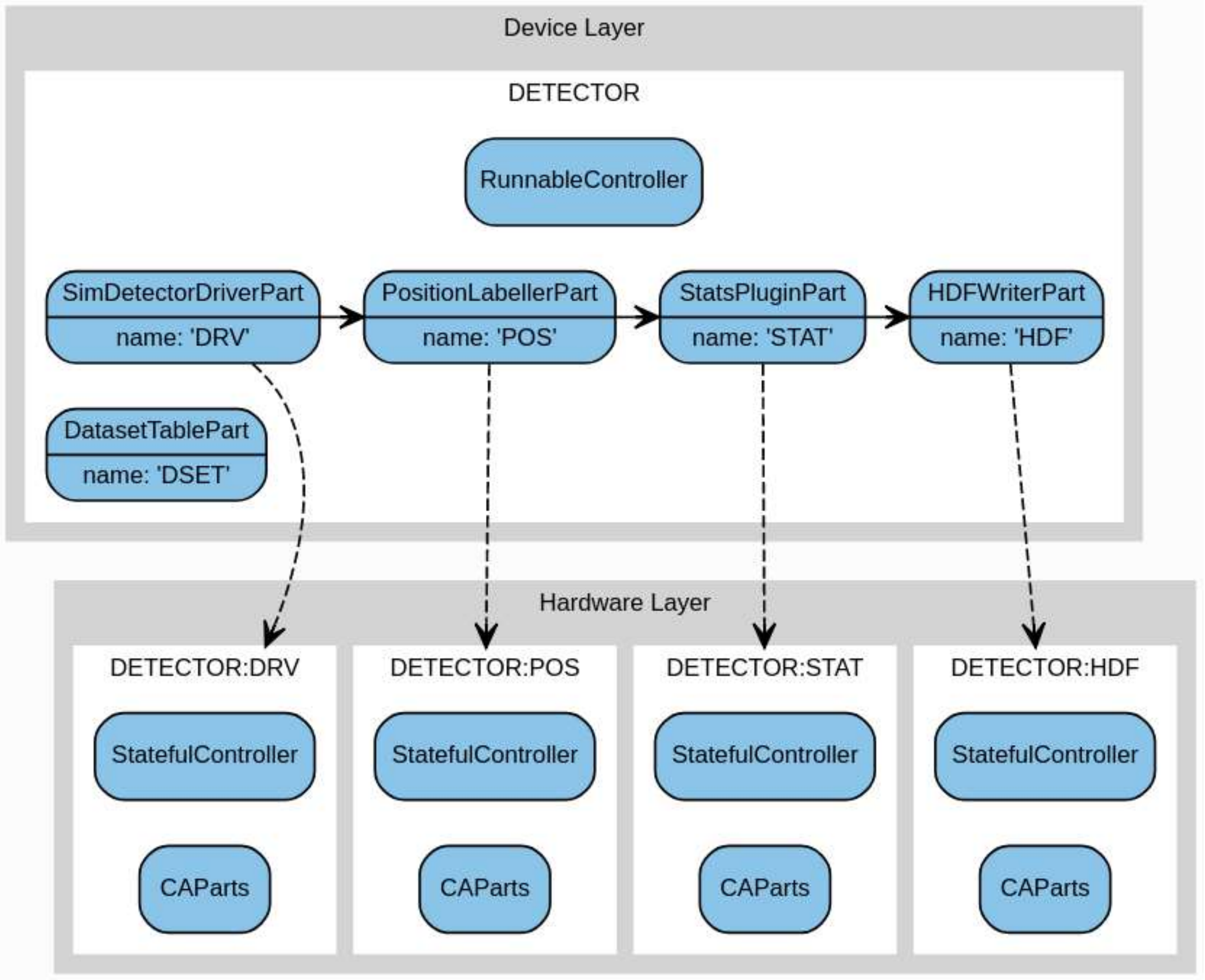
1) simDetector driver creates the NDArrays, each with a unique ID

2) NDPosPlugin tags each array with attributes that define its position within the dataset

3) NDPluginStats attaches statistics calculated from the data to each NDArray

4) NDFileHDF5 plugin writes the data to disk, using the attached attributes

AD Process Structure



Process Definition

demo/DEMO-AREADETECTOR.yaml (Part I)

```
- builtin.defines.cmd_string:
    name: hostname
    cmd: hostname –s

- builtin.defines.export_env_string:
    name: EPICS_CA_SERVER_PORT
    value: 6064

- builtin.defines.export_env_string:
    name: EPICS_CA_REPEATER_PORT
    value: 6065

- builtin.defines.string:
    name: config_dir
    value: /tmp
```

- ➢ Run the specified shell command and stores the result
- ➢ Export environment variables needed to talk to EPICS
- ➢ Define a $(config_dir) variable and set it to /tmp

Process Definition Cont..

demo/DEMO-AREADETECTOR.yaml (Part II)

```
- demo.blocks.motion_block:
   mri: $(hostname)-ML-MOT-01
   config_dir: $(config_dir)
- demo.blocks.detector_block:
   mri: $(hostname)-ML-DET-01
   config_dir: $(config_dir)
   label: Interference detector
- ADSimDetector.blocks
          .sim_detector_runnable_block:
   mri_prefix: $(hostname)-ML-DET-02
   config_dir: $(config_dir)
   pv_prefix: $(hostname)-AD-SIM-01
   label: Ramp detector
   drv_suffix: CAM
```

- Instantiate the child motion and detector blocks
- Instantiate a *sim_detector_runnable_block*
- Provide PV name information

Detector Device Block

- Instantiates a *RunnableController*
- Plus one block and one part for each element in the hardware layer (driver and plugins). For example:

```
- ADCore.blocks.stats_plugin_block:
    mri: $(mri_prefix):STAT
    prefix: $(pv_prefix):STAT
- ADCore.parts.StatsPluginPart:
    name: STAT
    mri: $(mri_prefix):STAT
```

- An include file pulls in commonly used items:

```
- ADCore.includes.filewriting_collection:
    pv_prefix: $(pv_prefix)
    mri_prefix: $(mri_prefix)
```

Hardware Blocks

- EPICS PV interface specified using Parts in the *ca* module
- These wrap up related PVs into methods and attributes:

```
- ca.parts.CALongPart:
    name: numImages
    description: Number of images to take if imageMode=Multiple
    pv: $(prefix):NumImages
    rbv_suffix: _RBV
- ca.parts.CAActionPart:
    name: stop
    description: Stop acquisition
    pv: $(prefix):Acquire
    value: 0
    wait: False
```

Preparing the Example

From the launcher, run the following and hit ‘Start IOC’:

Utilities -> GDA AreaDetector Simulation

```
./malcolm/imalcolm.py
malcolm/modules/demo/DEMO-AREADETECTOR.yaml
```

```
>>> from scanpointgenerator import LineGenerator, CompoundGenerator
>>> scan = self.block_view(“<hostname>-ML-SCAN-01")
>>> yline = LineGenerator("y", "mm", -1, 0, 6)
>>> xline = LineGenerator("x", "mm", 4, 5, 5, alternate=True)
>>> generator = CompoundGenerator([yline, xline], [], [], duration=0.5)
>>> scan.configure(generator, fileDir="/tmp")
```

Running the Example

- Query the scan block's datasets:

```
>>> from annotypes  import  json_encode
>>> print(json_encode(scan.datasets.value ,indent=4))
```

  – Note there are now datasets from both detectors, as well as the motor demand positions

- Monitor one of the datasets from a new terminal and start the scan from the Malcolm terminal:

```
h5watch /tmp/INTERFERENCE.h5/entry/uid
>>> scan.run()
```

Unique IDs

When the scan is finished:

- First reset the scan to close the files:

```
>>> scan.reset()
```

- Print the file contents for the RAMP detector:

```
h5dump -n /tmp/RAMP.h5
```

- Then look at the UniqueID dataset:

```
h5dump -d /entry/NDAttributes/NDArrayUniqueId
/tmp/RAMP.h5
```

Unique IDs cont.

Notice the ‘snake scan’ ordering of the frames:

```
DATASET "/entry/NDAttributes/NDArrayUniqueId" {
  DATATYPE  H5T_STD_I32LE
  DATASPACE  SIMPLE { ( 6, 5, 1, 1 ) / ( H5S_UNLIMITED, H5S_UNLIMITED, 1, 1) }
  DATA {
  (0,0,0,0): 1,
  (0,1,0,0): 2,
  (0,2,0,0): 3,
  (0,3,0,0): 4,
  (0,4,0,0): 5,
  (1,0,0,0): 10,
  (1,1,0,0): 9,
  (1,2,0,0): 8,
  (1,3,0,0): 7,
  (1,4,0,0): 6,
  ……….
```

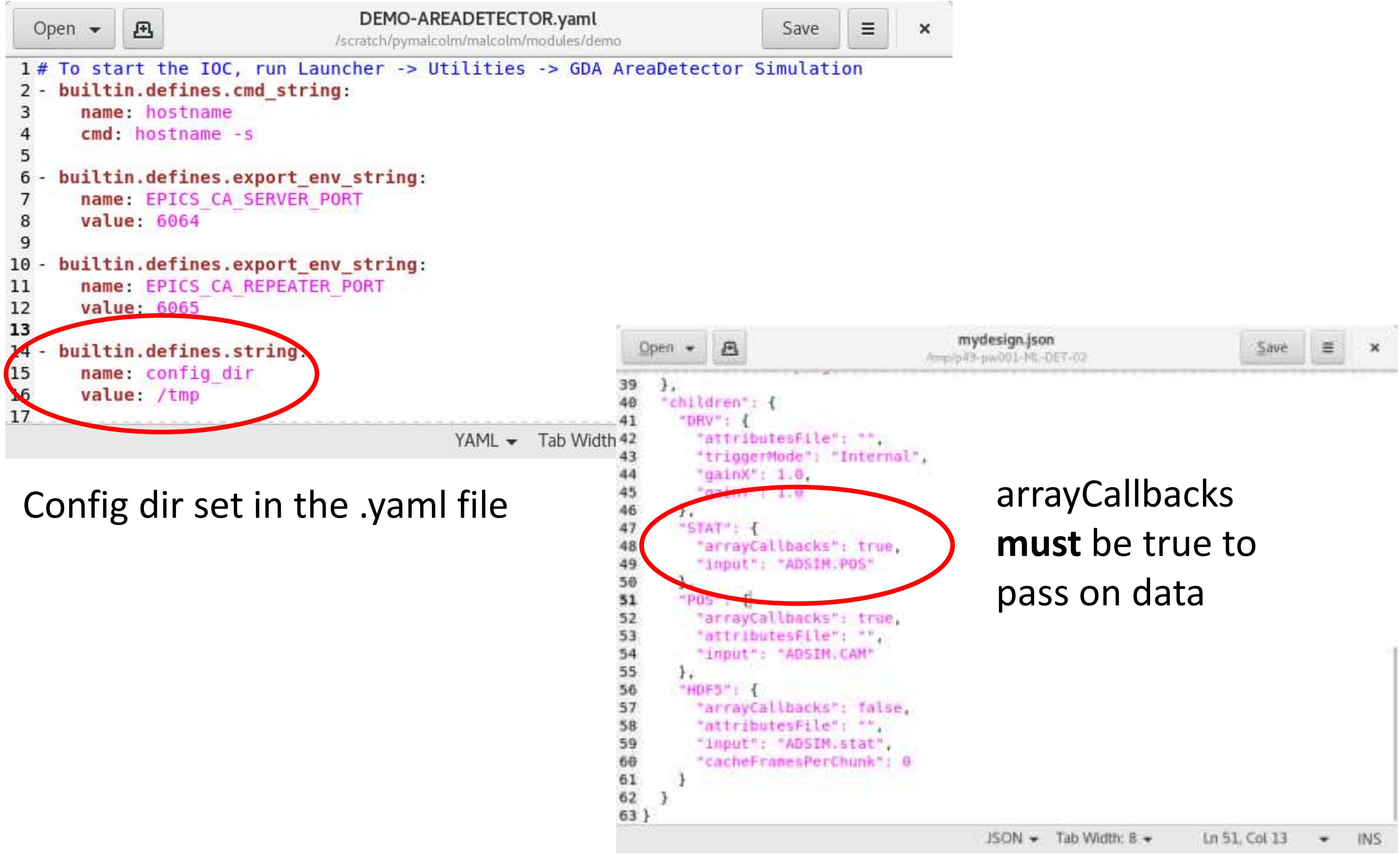
First row written left-to-right

Second row written right-to-left

Designs

- **Recall:** Designs describe the layout of a device block and the settings of its child blocks
- They allow plugin chains to be application specific
- They are saved as JSON files in the config_dir
- Use the *config* tag when defining the attribute in the Part to specify that it should be saved
- All writeable CAParts are tagged as *config* attributes by default
- This can be disabled by the class author

More on Designs



```yaml
# To start the IOC, run Launcher -> Utilities -> GDA AreaDetector Simulation
- builtin.defines.cmd_string:
    name: hostname
    cmd: hostname -s

- builtin.defines.export_env_string:
    name: EPICS_CA_SERVER_PORT
    value: 6064

- builtin.defines.export_env_string:
    name: EPICS_CA_REPEATER_PORT
    value: 6065

- builtin.defines.string:
    name: config_dir
    value: /tmp

```

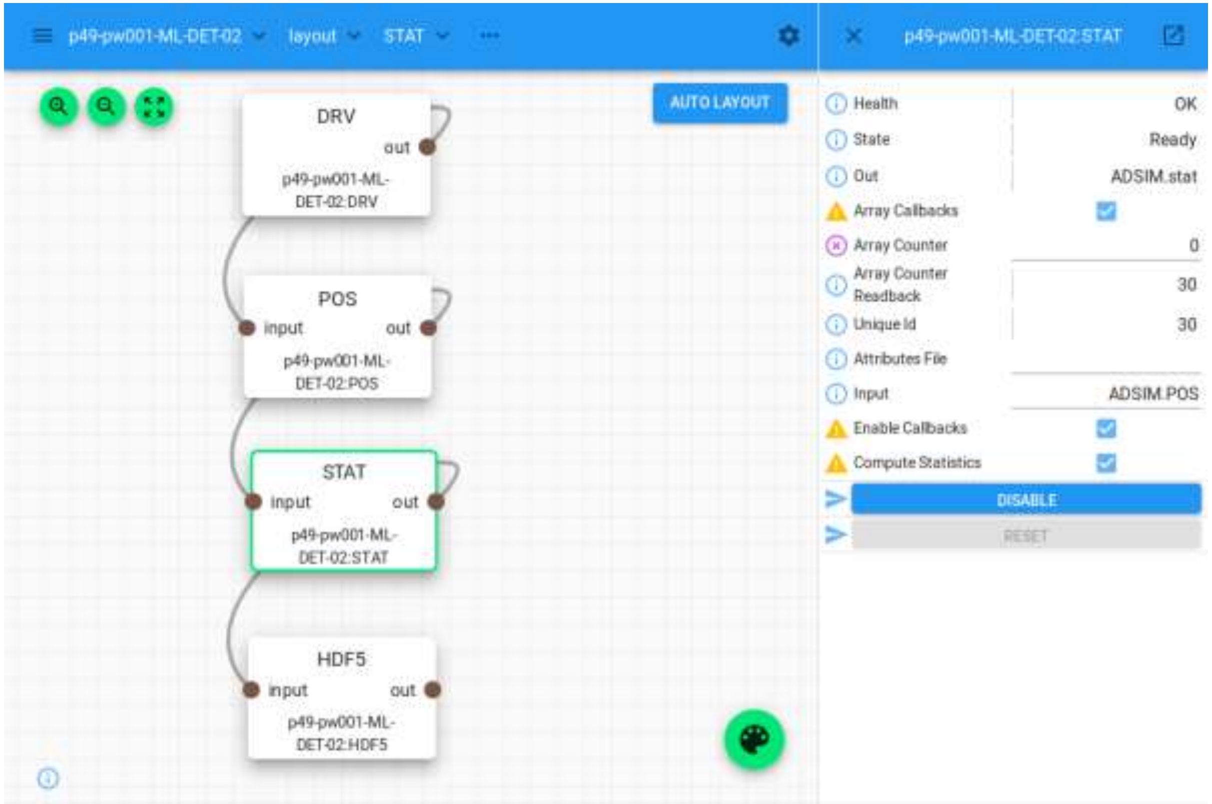
Config dir set in the .yaml file

```json
  },
  "children": {
    "DRV": {
      "attributesFile": "",
      "triggerMode": "Internal",
      "gainX": 1.0,
      "gainY": 1.0
    },
    "STAT": {
      "arrayCallbacks": true,
      "input": "ADSIM.POS"
    },
    "POS": {
      "arrayCallbacks": true,
      "attributesFile": "",
      "input": "ADSIM.CAM"
    },
    "HDF5": {
      "arrayCallbacks": false,
      "attributesFile": "",
      "input": "ADSIM.stat",
      "cacheFramesPerChunk": 0
    }
  }
}
```

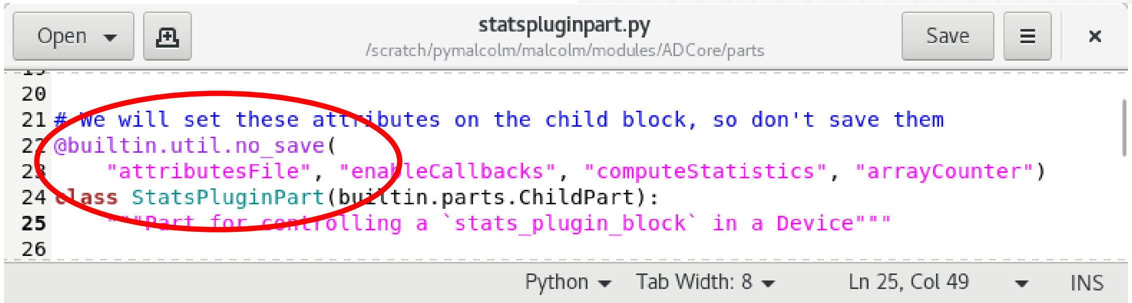
arrayCallbacks **must** be true to pass on data

More on Designs

Settings (e.g. plugin chain wiring) are saved by the *design*



Health	OK
State	Ready
Out	ADSIM.stat
Array Callbacks	☑
Array Counter	0
Array Counter Readback	30
Unique Id	30
Attributes File	
Input	ADSIM.POS
Enable Callbacks	☑
Compute Statistics	☑



```python

# We will set these attributes on the child block, so don't save them
@builtin.util.no_save(
    "attributesFile", "enableCallbacks", "computeStatistics", "arrayCounter")
class StatsPluginPart(builtin.parts.ChildPart):
    """Part for controlling a `stats_plugin_block` in a Device"""

```

Some plugin **attributes** are set during *configure()* so are excluded from the design

Template Designs

- Read-only designs provided by Malcolm
- Used as starting points for applications
- Named with the prefix template_
- Example: template_software_triggered:
  - Sets up the plugin chain correctly
  - Configures default trigger mode, gains etc.

Scan Level Designs

- Device blocks have designs for different scenarios
- Scan block design specifies the required combination of device block designs

```
"children": {
  “DETECTOR1": {
   "design": " template_software_triggered.json "
  },
  “DETECTOR2": {
   "design": "template_software_triggered.json"
  },
  "MOTOR": {
   "design": “hkl_geometry.json"
  }
```

Initial Designs

- Both *Manager* and *Runnable* Controllers can take an *initial_design* parameter
- **Device layer** blocks load their initial_design when Malcolm starts up
  - Warning: this means EPICS PVs may be written to!
- **Scan layer** blocks load their childrens' designs at the beginning of every scan

  https://pymalcolm.readthedocs.io/en/latest/tutorials/areadetector.html#scan-block-design

Practical Exercises

6. Experiment with the ‘pause and rewind’ feature to redo part of the scan in the Detector demo. What happens to the UniqueId dataset?
   - Hint: set the ‘last good step’ attribute, then press ‘pause’