Hardware Triggered Scanning: Scanning Components

Philip Taylor, Emma Arandjelovic

Observatory Sciences Limited

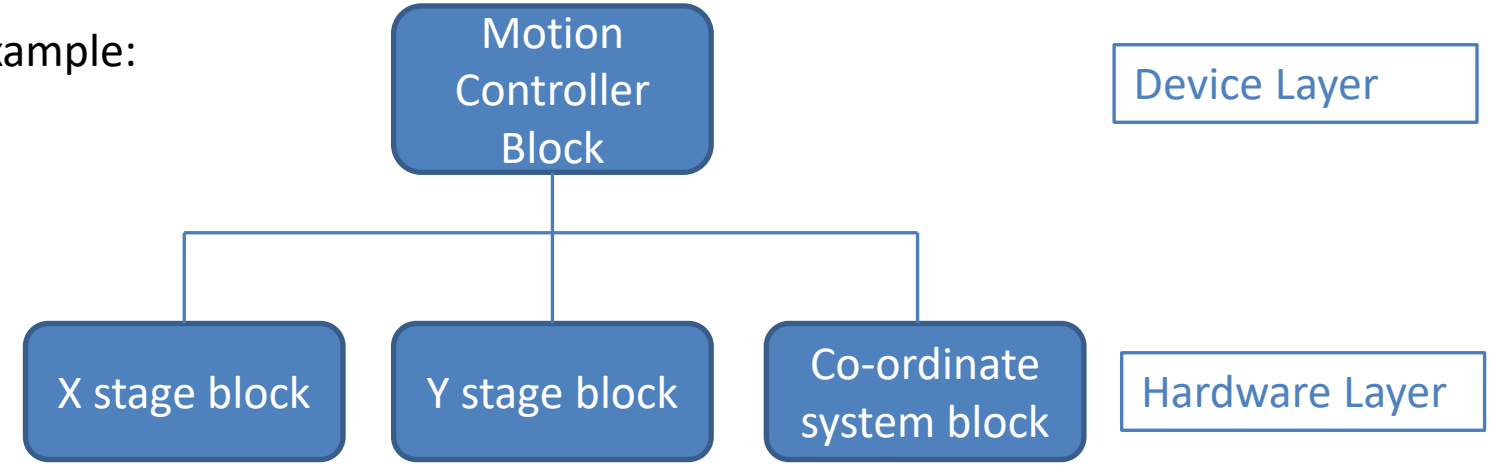
Overview

- To perform scans, we need to control complex devices such as motion controllers and detectors
- This topic will cover how these devices are configured by Malcolm
- We will use a simulated motion controller and detector to see how this works in practice
- We will also look at how to generate scan trajectories

Block Structure

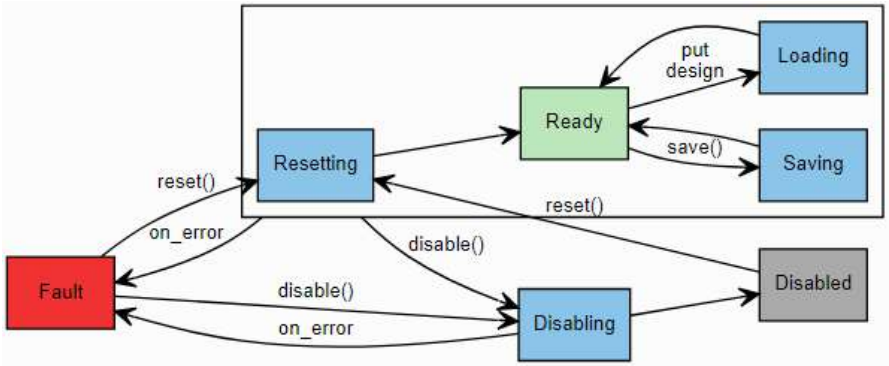
- Blocks like we have created so far make up the lowest level of blocks, known as the *Hardware Layer*
- The *Device Layer* is a higher level of blocks that synchronise a number of child blocks to create an interface to a device e.g. a detector or motor controller

Example:



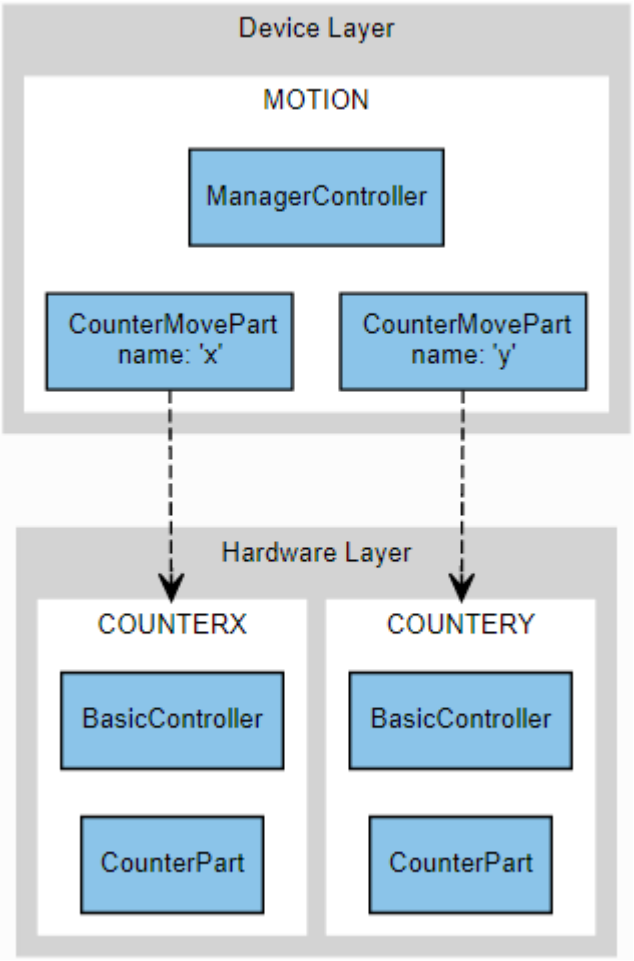
Device Blocks

- Device blocks instantiate one or more child blocks
- A controller aimed at managing child blocks is called a *ManagerController*
- This provides:
  - A *layout* of child blocks
  - A mechanism to enable / disable child blocks
  - A mechanism to load / save designs (saved configurations)
- State machine:



Motion Demo

- Two *Counter* blocks in the hardware layer simulate motor axes (X and Y)
- A device block called *Motion* simulates a motor controller with *XMove* and *YMove* methods
- The *Counter* blocks are constructed with *BasicControllers*
- The *Motion* block is constructed with a *ManagerController*



See the Motion tutorial:
https://pymalcolm.readthedocs.io/en/latest/tutorials/motion.html

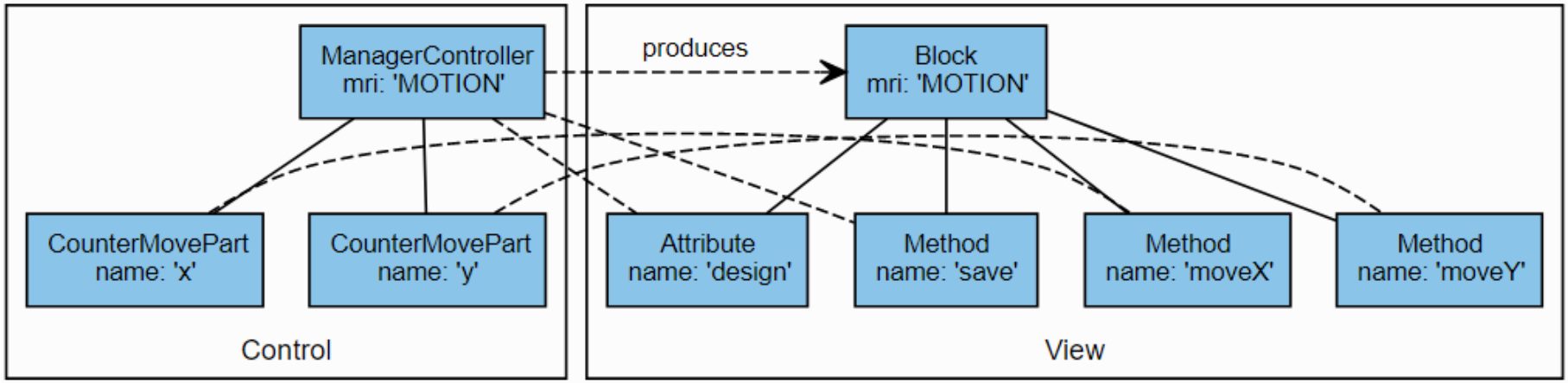
Motion Block Definition

demo/blocks/motion_block.yaml (extract):

```
- builtin.controllers.ManagerController:
    mri: $(mri)
    config_dir: $(config_dir)
    description: $(docstring)

- demo.blocks.counter_block:
    mri: $(mri):COUNTERX

- demo.blocks.counter_block:
    mri: $(mri):COUNTERY

- demo.parts.CounterMovePart:
    name: x
    mri: $(mri):COUNTERX

- demo.parts.CounterMovePart:
    name: y
    mri: $(mri): COUNTERY
```

- Use a *ManagerController* to construct the block
  - This takes an additional parameter to specify a directory for saving configurations
- Instantiate two *Counter* blocks
- Instantiate two *MotorMove* parts, one per block

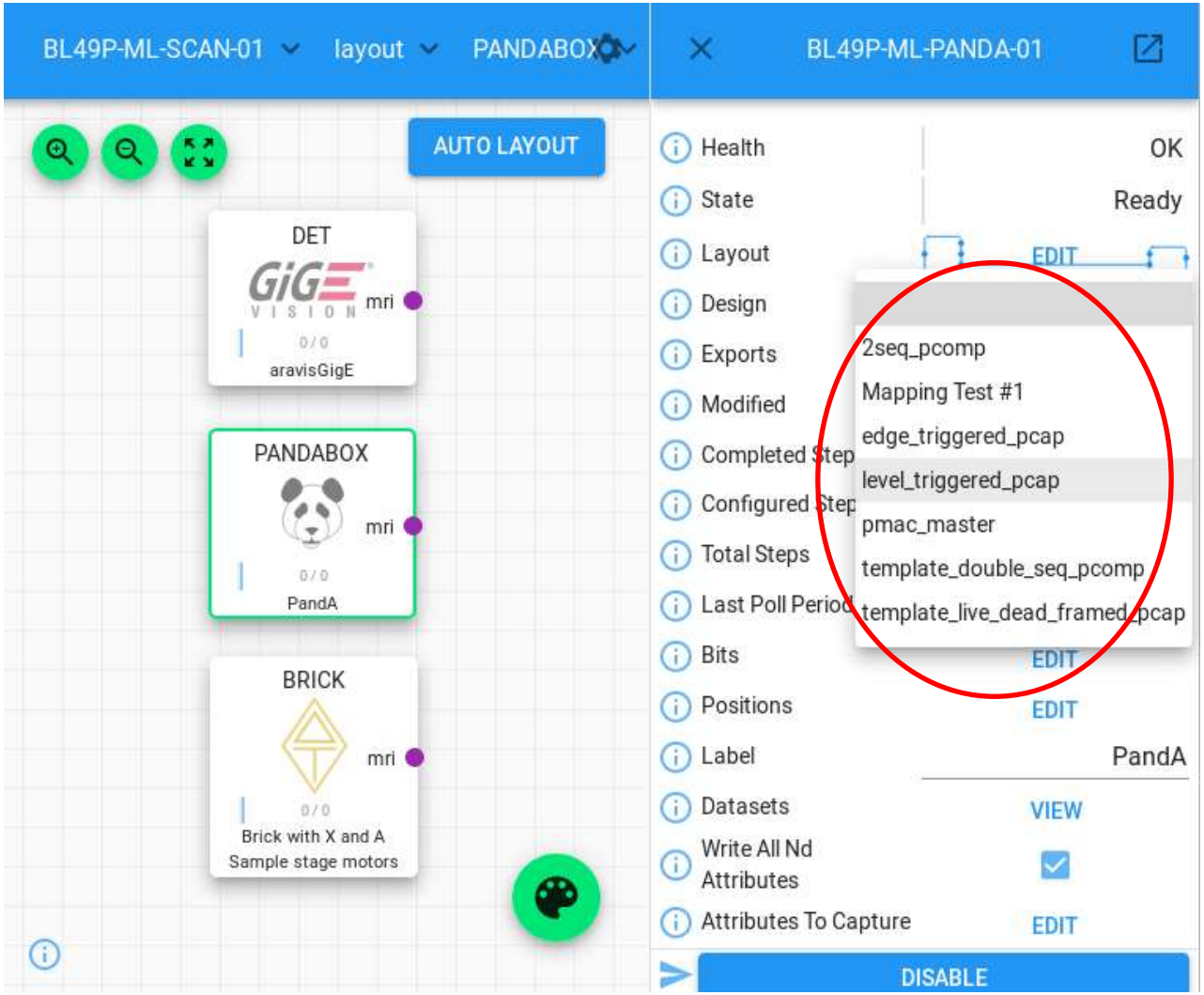
Motion Example Structure



Notice the *design* attribute and *save* method are contributed by the *ManagerController*

Designs

Recall: Designs allow you to save the layout of a device block and the attributes of its child blocks



Designs continued

- Which attributes are saved?
- Recall the attribute definition in our *Counter* part:

```
self.counter = NumberMeta(
        "float64", "The current value of the counter",
        tags=[config_tag(), Widget.TEXTINPUT.tag()])
```

- Tags contain important metadata:
  - Widget tags - choose which widget to display it with
  - Group tags - create an expander around related attributes
  - Port tags - create the visual connections between blocks
  - Config tags - turn on saving/loading

See: https://pymalcolm.readthedocs.io/en/latest/reference/tags.html

How to Create a Device Part

1. Subclass *builtin.parts.ChildPart*
   - This provides access to the child blocks
2. Create annotypes for any variables we need
3. Create an *__init__()* method:
   - Call *super().__init__()*
4. Create a *setup()* method:
   - Call *super().setup()*
   - Register any additional methods and attributes
5. Implement the methods to perform the business logic
   - Use the passed in *context* to access the child block

Motor Move Part Definition

demo/parts/countermovepart.py (extract):

```
with Anno("The demand value to move our
  counter motor to"):
    ADemand = float

@builtin.util.no_save(“counter”)
class CounterMovePart(builtin.parts.ChildPart):
  def __init__(self, name, mri):
    # type: (APartName, builtin.parts.AMri) -> None
    super(CounterMovePart,  self).__init__(name,
        mri, initial_visibility=True, stateful=False)
  def setup(self, registrar):
    # type: (PartRegistrar) -> None
    super(CounterMovePart, self).setup(registrar)
    registrar.add_method_model(self.move,
        self.name + “Move”, needs_context=True)
```

- Create an Annotype for the demand position
- Define attributes in the child block not to be saved
- Call super().__**init**__()
- Register the method model in **setup**, ensuring the name is unique for each part
- Tell Malcolm the method needs a *Context* object for accessing the child block

Part Definition Continued

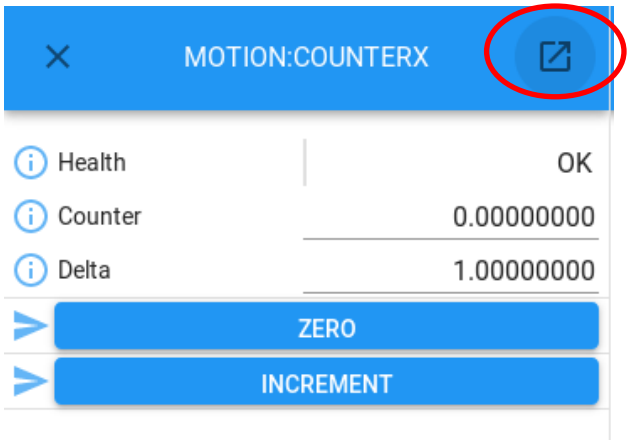
demo/parts/countermovepart.py (continued):

```
@add_call_types
def move(self, context, demand):
    # type: (builtin.hooks.AContext, ADemand) -> None
    child = context.block_view(self.mri)
    child.counter.put_value(demand)
```

1. Use the passed in context as well as the MRI to create a Block view of the *Counter* child Block
2. Set the Block's *counter* attribute to the new demand

Running the Motion Demo

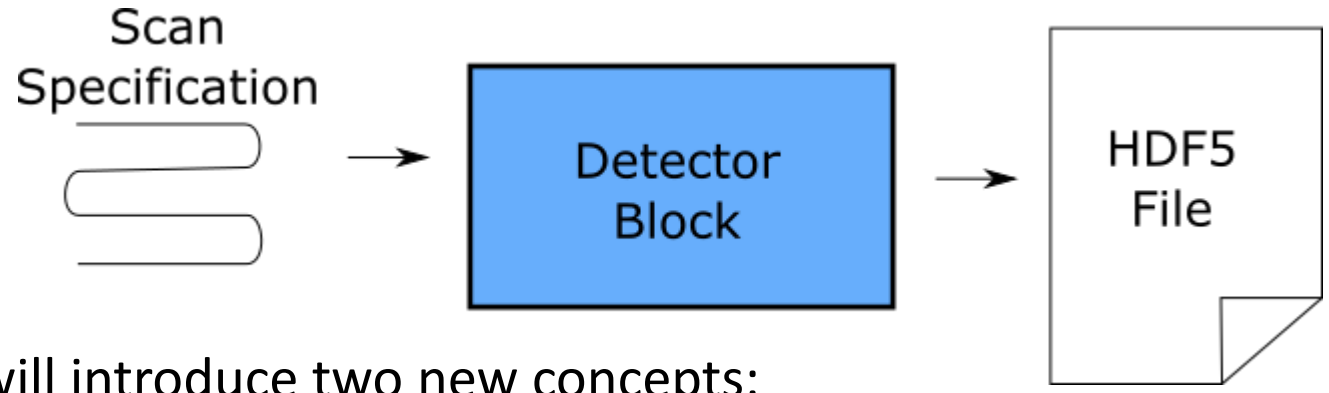
./malcolm/imalcolm.py
malcolm/modules/demo/DEMO-MOTION.yaml



1. Open http://localhost:8008/
2. Select the MOTION:COUNTERX block and ‘tear-off’
3. Repeat for MOTION:COUNTERY
4. Select the MOTION root block and request new values for X and Y
5. Watch the counter blocks changing value
6. Practise saving and loading designs with different *delta* values

Detector Demo

- A dummy detector that takes a scan specification and writes data to an HDF5 file



- This will introduce two new concepts:
  1. *RunnableController* – provides methods and attributes relating to the different steps of a scan
  2. *ScanPointGenerator* – used to generate n-dimensional scan paths

See the Detector tutorial:
https://pymalcolm.readthedocs.io/en/latest/tutorials/detector.html

Runnable Device Blocks

- A ‘Runnable’ device block understands what a scan is
- It has a state machine that implements the scan process
- This is controlled by the user via two important methods:
  - configure()

    Configure all child blocks according to the supplied parameters
  - run()

    Start all child blocks running, provide status monitoring and periodic actions

- These are contributed to the block by a *RunnableController*

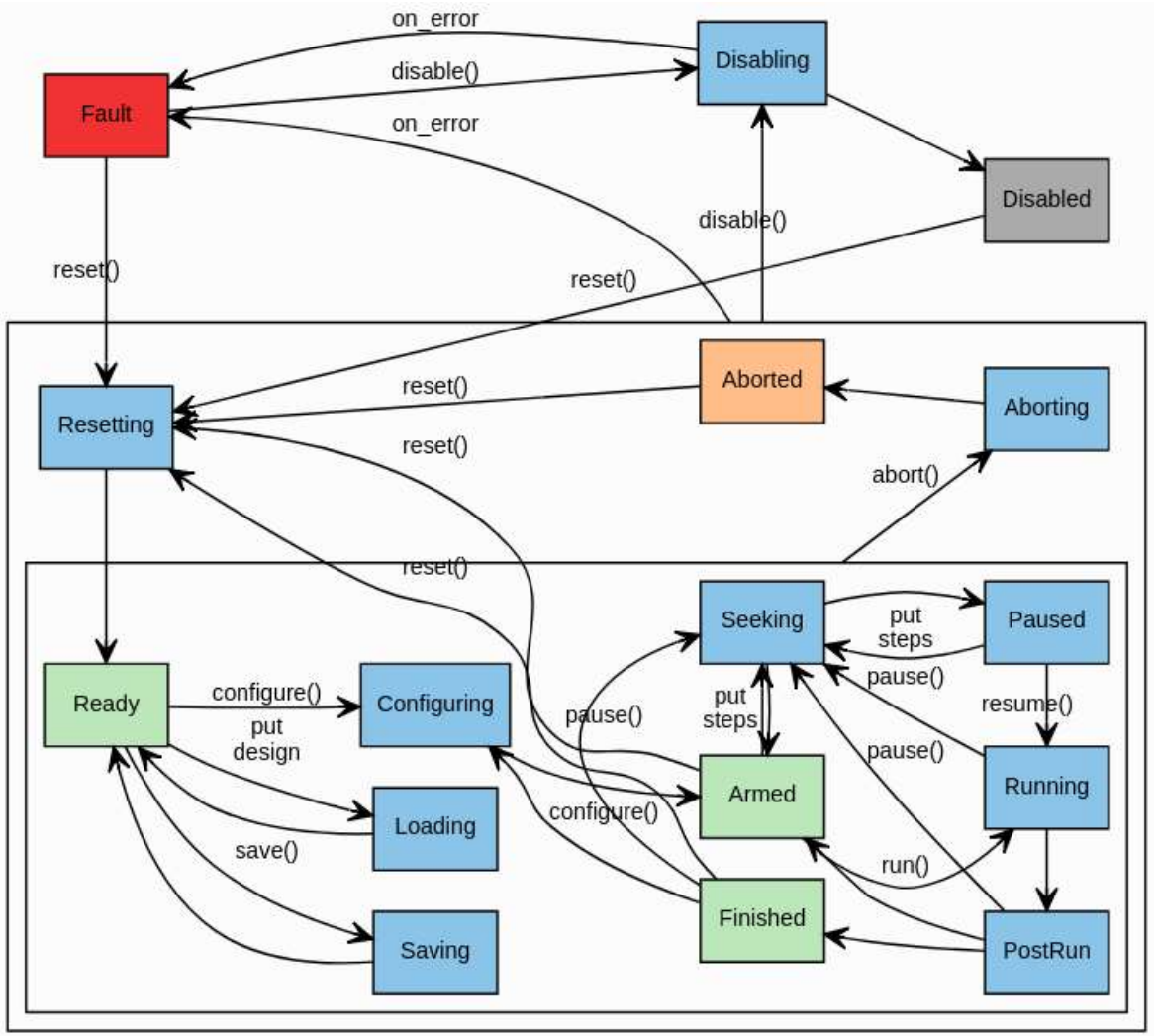
Runnable Controller Details

- Inherits from *ManagerController*
- Adds to the *ManagerController* state machine to include the different steps of a scan
- Additional attributes:
  - completedSteps
  - configuredSteps
  - totalSteps
- Controllers provide *hooks*, which Parts can register with to run the device specific logic on state transitions

Runnable Controller States

Hooks are run on state transitions

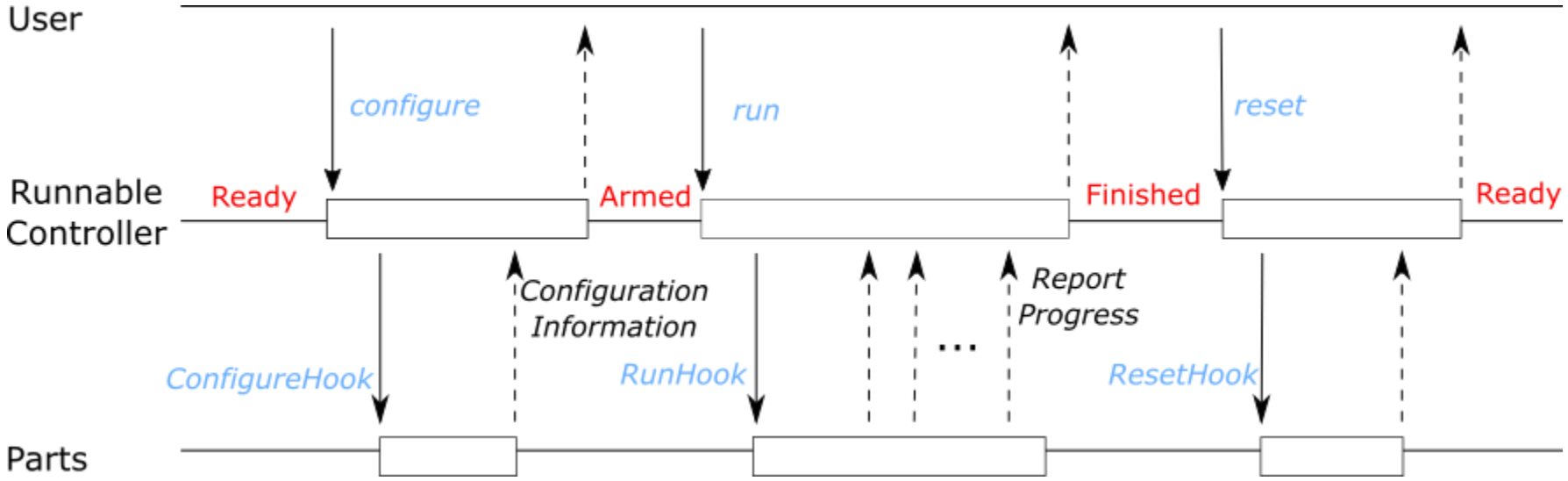
See: https://pymalcolm.readthedocs.io/en/latest/reference/statesets.html#statesets



Hooks Overview

- Functions registered to hooks are called automatically by the controller
- All functions hooked to the same hook are called concurrently
- Examples:
  - ConfigureHook (part, context, completed_steps, steps_to_do, part_info, generator, axesToMove, **kwargs):
    - Called at configure() to setup a child block
  - RunHook (part, context, **kwargs):
    - Called at run() to start all children running
  - ResetHook(part)
    - Called at reset() to return to Ready state

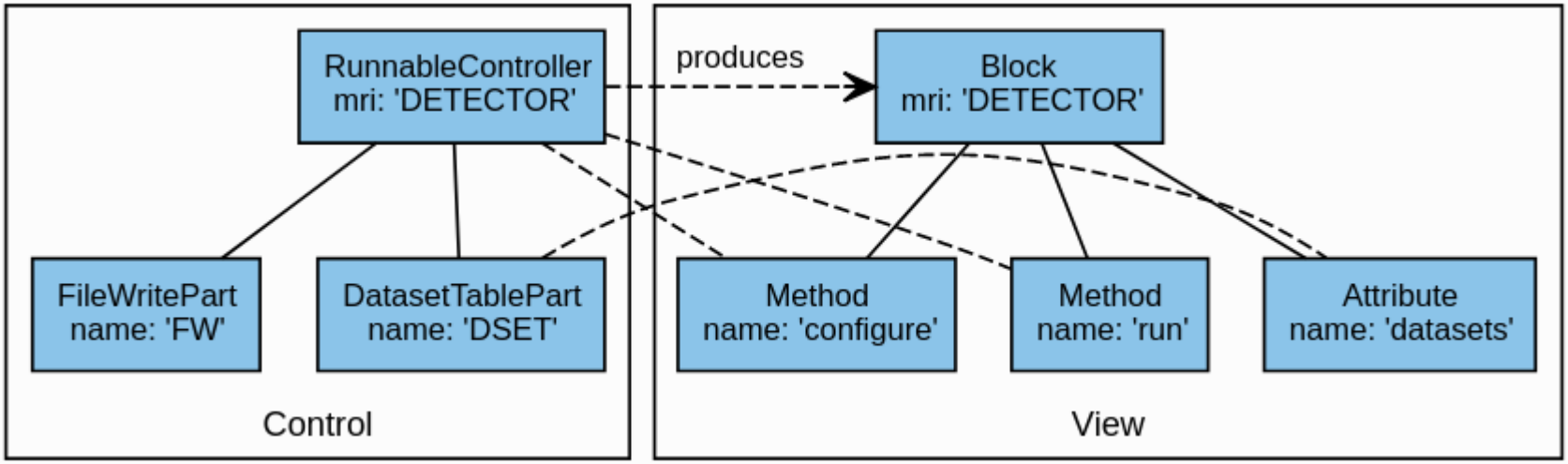
Runnable Device Sequence Diagrams



Note: There are more hooks than are shown here, see https://pymalcolm.readthedocs.io/en/latest/reference/statesets.html for a complete list of hooks.

Detector Demo Structure

- The *DETECTOR* block is constructed with a *RunnableController*
- A *FileWritePart* writes the HDF5 file
  - *Configure hook takes a scan point generator and initialises the scan*
  - *Run hook iterates through the generator points and writes the data*
- A *DatasetTablePart* collects info (location, size etc.) about the data and exposes it in the *datasets* attribute



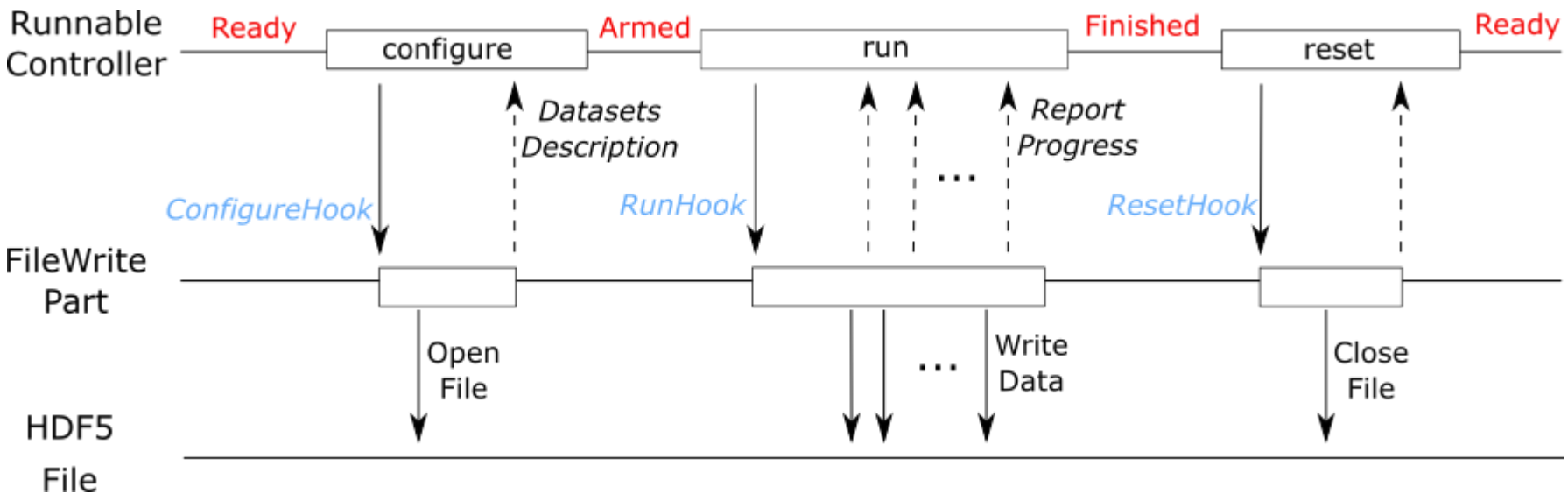
Detector Block Definition

demo/blocks/detector_block.yaml (extract):

```
- builtin.controllers.RunnableController:
    mri: $(mri)
    config_dir: $(config_dir)
    description: $(docstring)

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

- scanning.parts.DatasetTablePart:
    name: DSET

- demo.parts.FileWritePart:
    name: FW
    width: 160
    height: 120
```

- Use a *RunnableController* to construct the block
- A *LabelPart* displays a title on the GUI
- A *DatasetTablePart* reports the datasets written
- A *FileWritePart* writes some dummy data to an HDF5 file

Detector Demo Sequence Diagram



This demo detector saves some generated data to an HDF5 file just as an example.

How to Create a Runnable Device Part

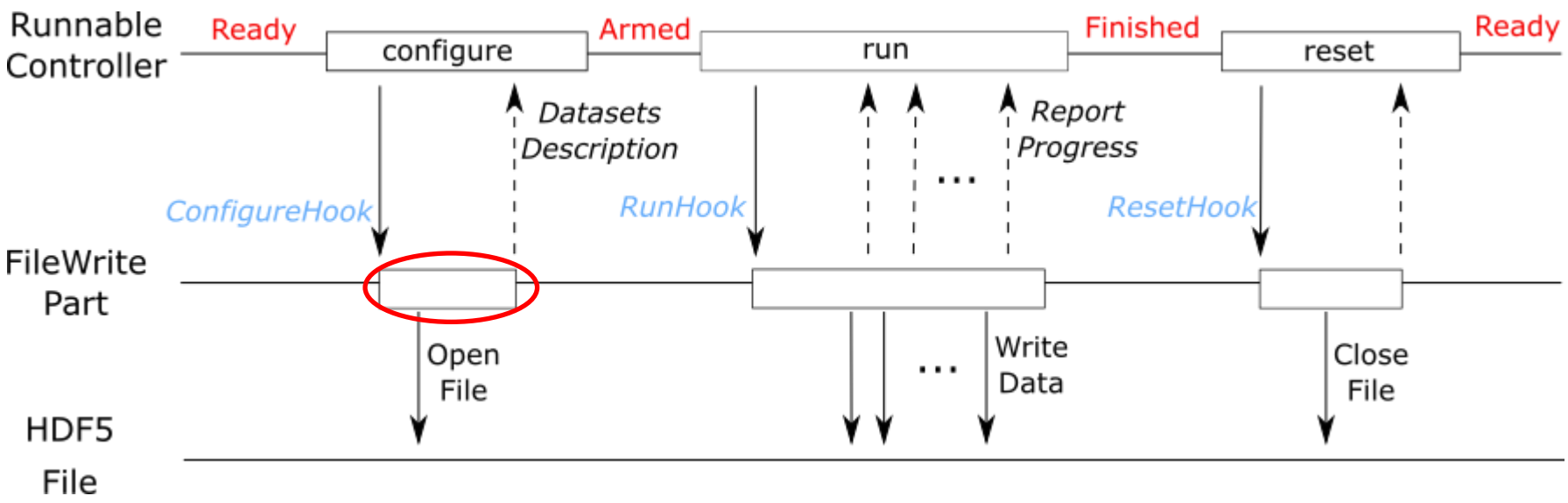
1. Subclass *builtin.parts.Part* or *builtin.parts.ChildPart*
2. Create an *__init__* method:
   - Call *super().__init__()*
3. Implement the *on_configure(*params)*, *on_run()* and *reset()* methods to perform the business logic
   - The *on_configure* method should prepare the scan
   - The *on_run* method should generate the data and report progress to the controller so it can update the block's *currentStep* attribute
   - The *reset* method should clean up after a scan, even if it failed.
4. Create a *setup* method:
   - Register any additional methods and attributes
   - Register any hooks with the controller
   - Inform the controller what extra parameters *configure* requires

File Write Part: init

demo/parts/filewritepart.py (extract):

```
class FileWritePart(Part):

    def __init__(self, name, width, height):
        # type: (APartName, AWidth, AHeight) -> None
        super(FileWritePart, self).__init__(name)
        # Store input arguments
        self._width = width
        self._height = height
        # The detector image we will modify for each image (0..255 range)
        self._blob = make_gaussian_blob(width, height) * 255
        # The HDF5 file we will write
        self._hdf = None            # type: h5py.File
        # Configure args and progress info
        self._generator = None  # type: scanning.hooks.AGenerator
        self._completed_steps = 0
        self._steps_to_do = 0
        # How much to offset unique id value from generator point (for rewind function)
        self._uid_offset = 0
```

Detector Demo Sequence Diagram



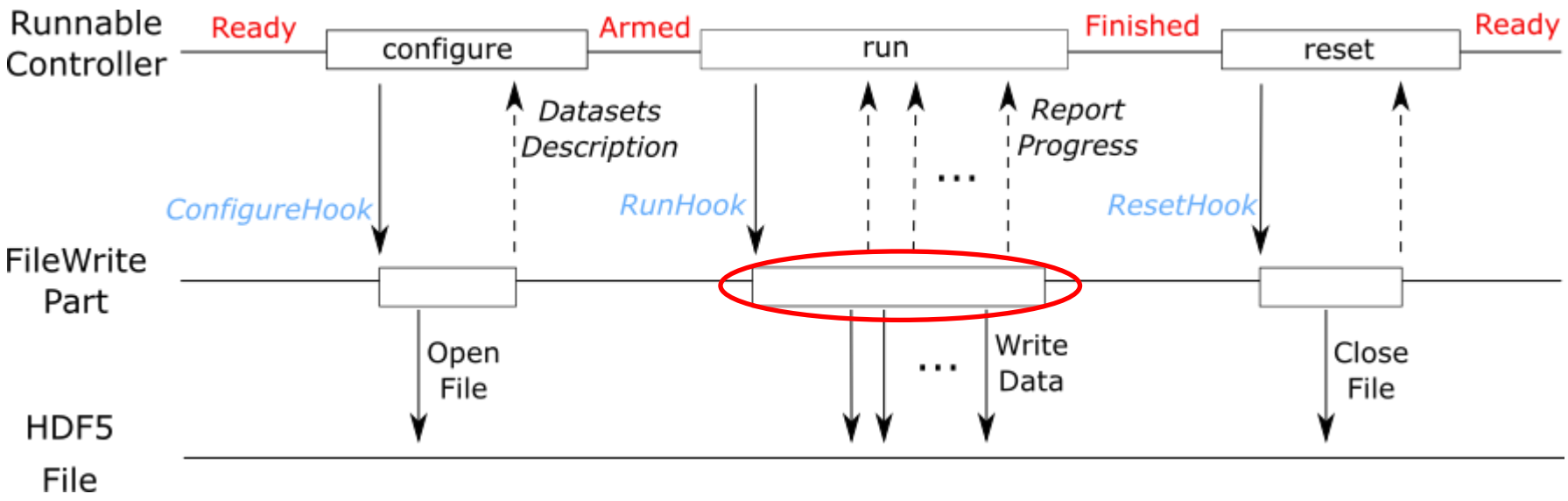
This demo detector saves some generated data to an HDF5 file just as an example.

File Write Part: on_configure

demo/parts/filewritepart.py (extract):

```
def on_configure(self,
                 completed_steps,      # type: scanning.hooks.ACompletedSteps
                 steps_to_do,          # type: scanning.hooks.AStepsToDo
                 generator,            # type: scanning.hooks.AGenerator
                 fileDir,              # type: scanning.hooks.AFileDir
                 formatName="det",     # type: scanning.hooks.AFormatName
                 fileTemplate="%s.h5", # type: scanning.hooks.AFileTemplate
                 ):
    # type: (...) -> scanning.hooks.UInfos
    # Store the parameters and create an empty HDF file
    self._generator = generator
    filename = fileTemplate % formatName
    filepath = os.path.join(fileDir, filename)
    self._hdf = self._create_hdf(filepath, generator)
    # Create the Info objects describing the two datasets.....
    infos = [scanning.infos.DatasetProducedInfo(name=...) ,...]
    return infos
```

Detector Demo Sequence Diagram



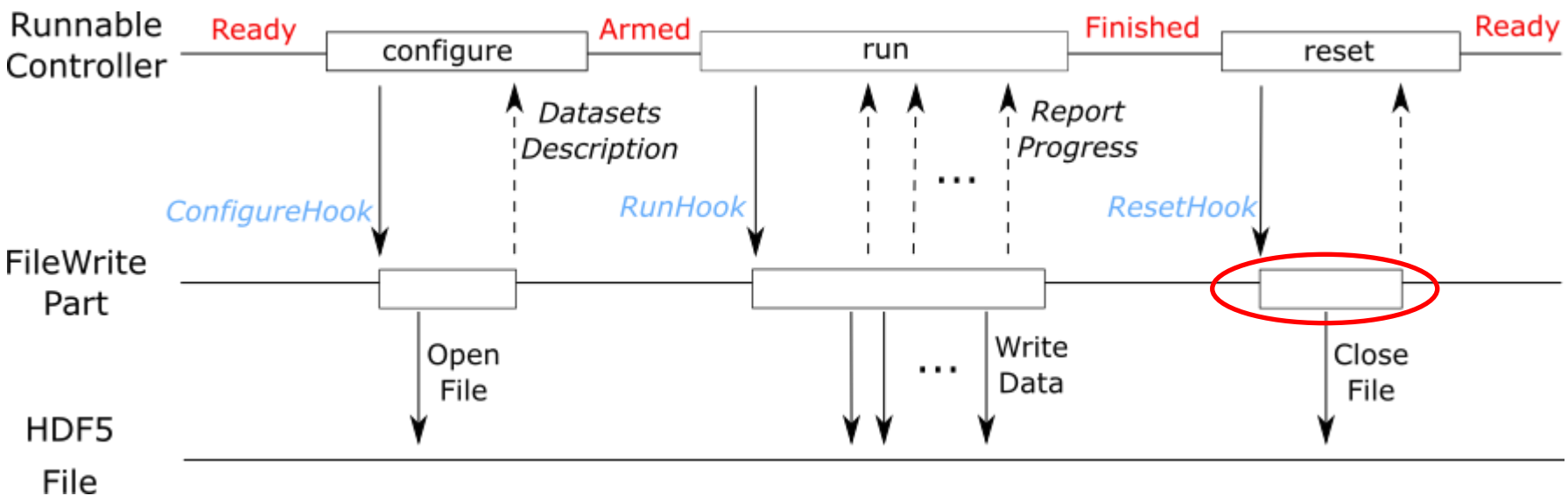
This demo detector saves some generated data to an HDF5 file just as an example.

File Write Part: on_run

demo/parts/filewritepart.py (extract):

```
def on_run(self, context):
    # type: (scanning.hooks.AContext) -> None
    """On `RunHook` record where to next take data"""
    end_of_exposure = time.time() + self._exposure # Start time so everything is relative
    last_flush = end_of_exposure
    for i in range(self._completed_steps, self._completed_steps + self._steps_to_do):
        point = self._generator.get_point(i)
        wait_time = end_of_exposure - time.time()
        context.sleep(wait_time) # Must use the context parameter to make it
                                 # an interruptible sleep so the scan can be aborted
        self.log.debug("Writing data for point %s", i)
        self._write_data(point, i)
        if time.time() - last_flush > FLUSH_PERIOD:
            last_flush = time.time()
            self._flush_datasets()
            end_of_exposure += point.duration
        self.registrar.report(scanning.infos.RunProgressInfo(i + 1))
    self._flush_datasets()
```

Detector Demo Sequence Diagram



This demo detector saves some generated data to an HDF5 file just as an example.

File Write Part: reset

demo/parts/filewritepart.py (extract):

```
def on_reset(self):
    # type: () -> None
    if self._hdf:
        self._hdf.close()
        self._hdf = None
```

File Write Part: setup

demo/parts/filewritepart.py (continued):

```
def setup(self, registrar):
    # type: (PartRegistrar) -> None
    super(FileWritePart, self).setup(registrar)
    # Hooks
    registrar.hook(scanning.hooks.ConfigureHook, self.on_configure)
    registrar.hook((scanning.hooks.PostRunArmedHook,
                    scanning.hooks.SeekHook), self.on_seek)
    registrar.hook(scanning.hooks.RunHook, self.on_run)
    registrar.hook((scanning.hooks.AbortHook,
                    builtin.hooks.ResetHook), self.on_reset)
    # Tell the controller to expose some extra configure parameters
    registrar.report(scanning.hooks.ConfigureHook.create_info(self.on_configure))
```

Additional Hooks

More examples:

- SeekHook
  - Used for pause/rewind functionality, to adjust the *completed_steps*
- PostRunArmedHook
  - Called at the end of run() when there are more steps to be run
    (e.g. when the scan is paused)
- AbortHook
  - Called at abort() to stop the current scan
- ResetHook
  - Called at reset() to reset all Parts to a known state

Info Objects

- An *Info* is an object created by a Part and passed to its parent Controller in one of two ways:
  - Returned from a hooked function
  - Reported via *PartRegistrar.report(info)*
- We report progress via *PartRegistrar.report(info)* in the method registered to the RunHook:

```
def on_run(self, context):
    …
    for i in range(…):
        …
        self.registrar.report(
            scanning.infos.RunProgressInfo(i+1))
```

Info Objects Continued

- We return a description of the dataset produced by the *Part* from the method registered to the *ConfigureHook*:

```
def on_configure(self, …):
    # type: (...) -> scanning.hooks.UInfos
    …
    infos = [scanning.infos.DatasetProducedInfo(name=…), …]
    return infos
```

- *DatasetProducedInfo* parameters:
  - **name** – Dataset name
  - **filename** – Filename relative to the fileDir we were given
  - **type** – What NeXuS dataset type it produces
  - **rank** – The rank of the dataset including generator dims
  - **path** – The path of the dataset within the file
  - **uniqueid** – The path of the UniqueID dataset within the file

Info Objects Continued

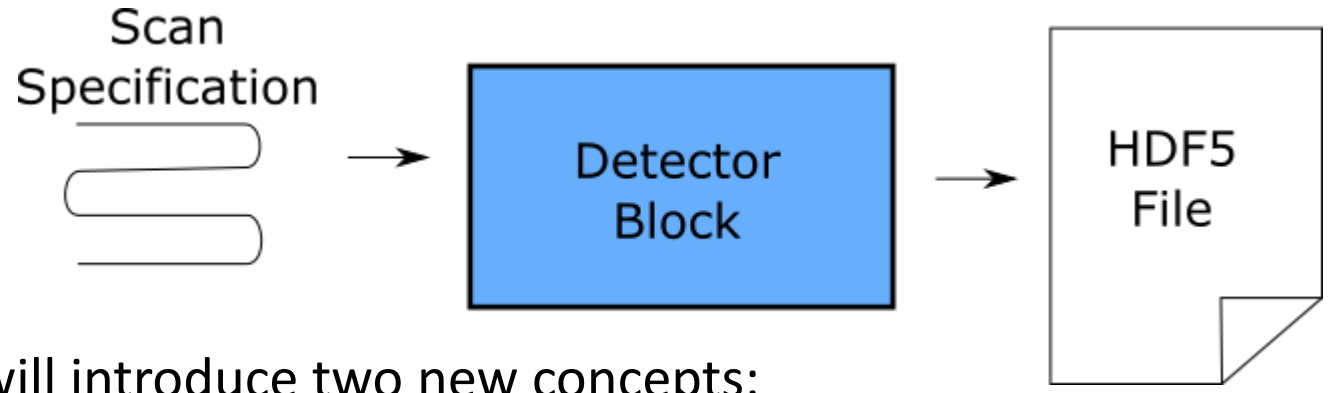
- We report additional arguments for hook functions in the *setup* method:

```
def setup(self, registrar):
    …
    registrar.report(
        scanning.hooks.ConfigureHook.create_info(self.configure))
```

- *create_info* scans the *on_configure* method for the arguments in addition to *completed_steps* and *steps_to_do:*
  - *generator*
  - *fileDir*
  - *formatName*
  - *fileTemplate*

Detector Demo

- A dummy detector that takes a scan specification and writes data to an HDF5 file



- This will introduce two new concepts:
  1. *RunnableController* – provides methods and attributes relating to the different steps of a scan
  2. *ScanPointGenerator* – used to generate n-dimensional scan paths

See the Detector tutorial:
https://pymalcolm.readthedocs.io/en/latest/tutorials/detector.html

Scan Point Generator

- To run a continuous scan, we need many parameters:
  - Demand positions for frame midpoint
  - Demand positions for frame start and end
  - Index at which to store the frame
  - Duration of the frame
- A Python module called *ScanPointGenerator* generates this data for a variety of scan types (e.g. line, spiral, …)
- A generator is passed to the *configure* method to setup the scan
- See https://scanpointgenerator.readthedocs.io

Scan Point Generator Example

class scanpointgenerator.LineGenerator
(axes, units, start, stop, size, alternate=False)
- Generate a line of equally spaced N-dimensional points

Parameters:

- axes (str/list(str)) – The scannable axes E.g. “x”
- units (str/list(str)) – The scannable units. E.g. “mm”
- start (float/list(float)) – The first position to be generated. e.g. 1.0
- stop (float/list(float)) – The final position to be generated. e.g. 5.0
- size (int) – The number of points to generate. E.g. 5
- alternate (bool) – Specifier to reverse direction

Preparing the Example

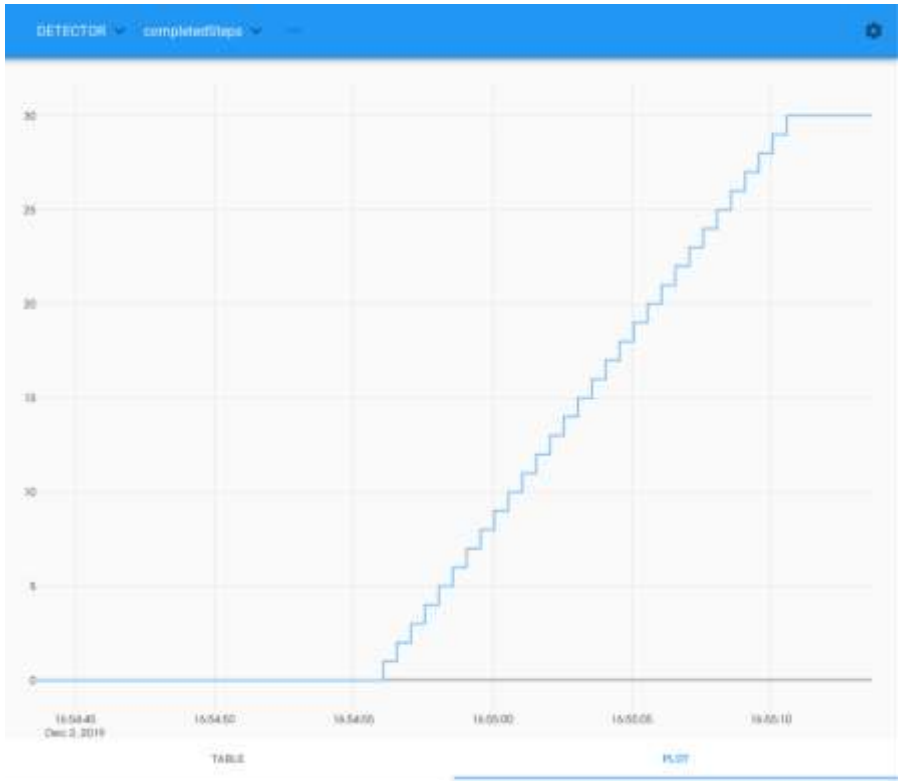
./malcolm/imalcolm.py malcolm/modules/demo/DEMO-DETECTOR.yaml

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

- Copy the JSON output to the clipboard

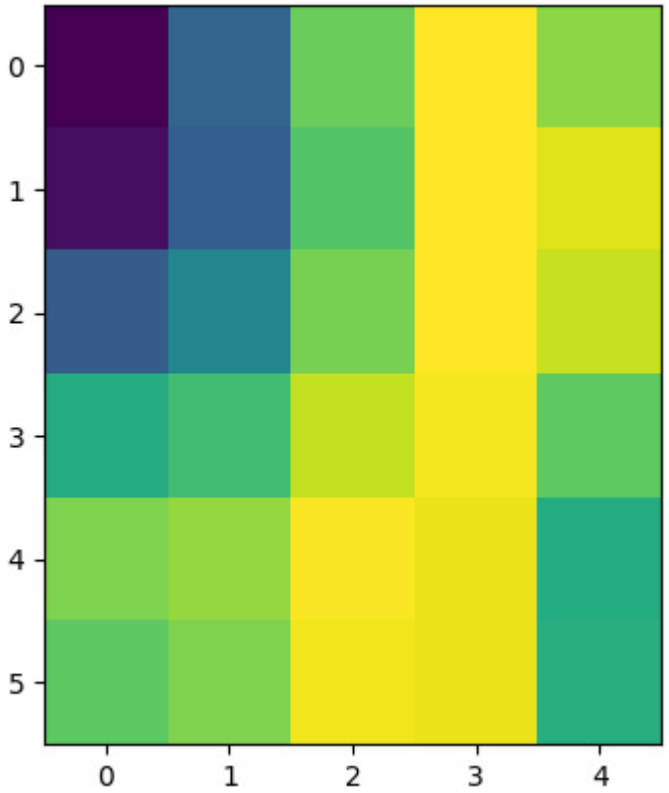
Running the Example

- Open http://localhost:8008/gui/DETECTOR
- Expand the *Configure* method and *Edit* the generator field to paste in the JSON code
- Set the *fileDir* to /tmp
- Click *Configure*
- Click *Run*
- Watch the *Completed Steps counter* increase



Checking the Results

```
>>> import h5py
>>> with h5py.File("/tmp/det.h5") as f:
....:     im = f["/entry/sum"][:]
....:
>>> im.shape
(6, 5, 1, 1)
>>> from pylab import imshow, show
>>> imshow(im[:, :, 0, 0])
<matplotlib.image.AxesImage at 0x7f11b090>
>>> show()
```



Recap: yaml chains

DEMO-DETECTOR.yaml

```yaml
# Create some Blocks
- demo.blocks.detector_block:
    mri: DETECTOR
    config_dir: /tmp

# Add a webserver
- web.blocks.web_server_block:
    mri: WEB
```

DEMO-DETECTOR.yaml | *detector_block.yaml

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

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

- builtin.parameters.int32:
    name: width
    description: Width of the produced image
    default: 160

- builtin.parameters.int32:
    name: height
    description: Height of the produced image
    default: 120

- demo.parts.FileWritePart:
    name: FW
    width: $(width)
    height: $(height)
```

DEMO-DETECTOR.yaml | detector_block.yaml | filewritepart.py

```python
class FileWritePart(Part):
    """Minimal interface demonstrating a file writing detector part"""
    def __init__(self, name, width, height):
        # type: (APartName, AWidth, AHeight) -> None
        super(FileWritePart, self).__init__(name)
        # Store input arguments
        self._width = width
        self._height = height
        # The detector image we will modify for each image (0..255 range)
        self._blob = make_gaussian_blob(width, height) * 255
        # The hdf file we will write
        self._hdf = None  # type: h5py.File
        # Configure args and progress info
        self._exposure = None
        self._generator = None  # type: scanning.hooks.AGenerator
        self._completed_steps = 0
        self._steps_to_do = 0
        # How much to offset uid value from generator point
        self._uid_offset = 0
```

Recap: yaml chains

DEMO-DETECTOR.yaml

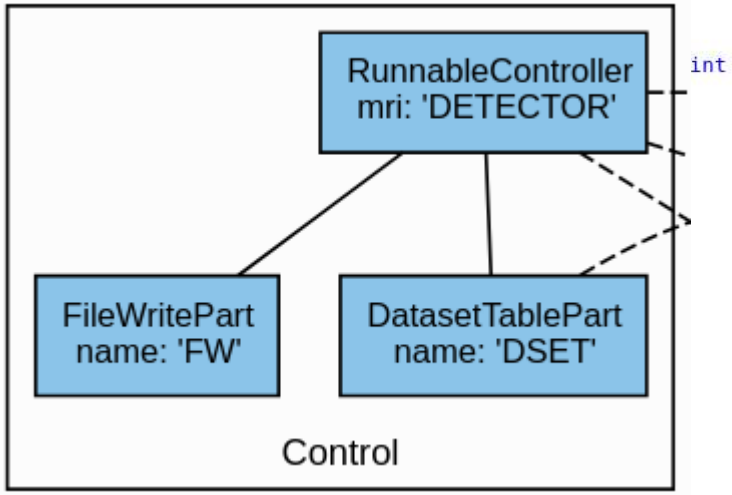
```yaml
# Create some Blocks
- demo.blocks.detector_block:
    mri: DETECTOR
    config_dir: /tmp

# Add a webserver
- web.blocks.web_server_block:
    mri: WEB
```

DEMO-DETECTOR.yaml | *detector_block.yaml

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

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

- builtin.parameters.int32:
    name: width
    description: Width of the produced image
    default: 160

- builtin.parameters.int32:
    name: height
    description: Height of the produced image
    default: 120

- demo.parts.FileWritePart:
    name: FW
    width: $(width)
    height: $(height)
```

DEMO-DETECTOR.yaml | detector_block.yaml | filewritepart.py

```python
class FileWritePart(Part):
    """Minimal interface demonstrating a file writing detector part"""
    def __init__(self, name, width, height):
        # type: (APartName, AWidth, AHeight) -> None
        super(FileWritePart, self).__init__(name)
        # Store input arguments
        self._width = width
        self._height = height
        # The detector image we will modify for each image (0..255 range)
        self._blob = make_gaussian_blob(width, height) * 255
        # The hdf file we will write
        self._hdf = None  # type: h5py.File
        # Configure args and progress info
        self._exposure = None
        self._generator = None  # type: scanning.hooks.AGenerator
```



Practical Exercises

4. Modify the Motion demo to add high and low soft limits (values to be provided in DEMO-MOTION.yaml). If a demand is requested out of this range, it should raise an error. Provide default values of 0, meaning soft limits are disabled.
   Hint 1: Start by modifying the DEMO-MOTION.yaml file and work your way down.
   Hint 2: Read the docs! See builtin parameter types from second page:
   https://pymalcolm.readthedocs.io/en/latest/tutorials/motion.html
   https://pymalcolm.readthedocs.io/en/latest/build/builtin/parameters_api.html

5. Modify the generator in the Detector demo to create a larger and faster scan.