SKA Tango Base Documentation

Release 0.17.0

NCRA India

Jun 08, 2023

CONTENTS

This package provides shared functionality and patterns for SKA TANGO devices.

CHAPTER

ONE

DEVELOPER GUIDE

1.1 Getting started

This page will guide you through the steps to writing a SKA Tango device based on the `ska-tango-base` package.

1.1.1 Prerequisites

It is assumed here that you have a subproject repository, and have set up your development environment. The `ska-tango-base` package can be installed from the SKAO repository:

```
me@local:~$ python3 -m pip install --extra-index-url https://artefact.skao.int/
↪repository/pypi/simple ska-tango-base
```

1.1.2 Basic steps

The recommended basic steps to writing a SKA Tango device based on the `ska-tango-base` package are:

1. Write a component manager.
2. Implement command class objects.
3. Write your Tango device.

1.1.3 Detailed steps

Write a component manager

A fundamental assumption of this package is that each Tango device exists to provide monitoring and control of some *component* of a SKA telescope. That *component* could be some hardware, a software service or process, or even a group of subservient Tango devices.

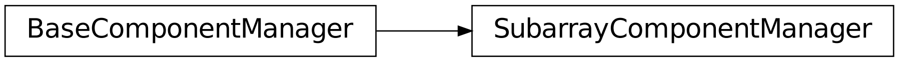
A *component manager* provides for monitoring and control of a component. It is *highly recommended* to implement and thoroughly test your component manager *before* embedding it in a Tango device.

For more information on components and component managers, see *Components and component managers*.

Writing a component manager involves the following steps.

1. **Choose a subclass for your component manager.** There are several component manager base classes, each associated with a device class. For example,

- If your Tango device will inherit from `SKABaseDevice`, then you will probably want to base your component manager on the `BaseComponentManager` class.
- If your Tango device is a subarray, then you will want to base your component manager on `SubarrayComponentManager`.



These component managers are abstract. They specify an interface, but leave it up to you to implement the functionality. For example, `BaseComponentManager`'s `on()` command looks like this:

```
def on(self):
  raise NotImplementedError("BaseComponentManager is abstract.")
```

Your component manager will inherit these methods, and override them with actual implementations.

Note: In addition to these abstract classes, there are also reference implementations of concrete subclasses. For example, in addition to an abstract `BaseComponentManager`, there is also a concrete `ReferenceBaseComponentManager`. These reference implementations are provided for explanatory purposes: they illustrate how a concrete component manager might be implemented. You are encouraged to review the reference implementations, and adapt them to your own needs; but it is not recommended to subclass from them.

2. **Establish communication with your component.** How you do this will depend on the capabilities and interface of your component. for example:
   - If the component interface is via a connection-oriented protocol (such as TCP/IP), then the component manager must establish and maintain a *connection* to the component;
   - If the component is able to publish updates, then the component manager would need to subscribe to those updates;
   - If the component cannot publish updates, but can only respond to requests, then the component manager would need to initiate polling of the component.
4. **Implement component monitoring.** Whenever your component changes its state, your component manager needs to become reliably aware of that change within a reasonable timeframe, so that it can pass this on to the Tango device.

   The abstract component managers provided already contain some helper methods to trigger device callbacks. For example, `BaseComponentManager` provides a `component_fault` method that lets the device know that the component has experienced a fault. You need to implement component monitoring so that, if the component experiences a fault, this is detected, and results in the `component_fault` helper method being called.

   For component-specific functionality, you will need to implement the corresponding helper methods. For example, if your component reports its temperature, then your component manager will need to

   1. Implement a mechanism by which it can let its Tango device know that the component temperature has changed, such as a callback;
   2. Implement monitoring so that this mechanism is triggered whenever a change in component temperature is detected.

5. **Implement component control.** Methods to control the component must be implemented; for example the component manager's `on()` method must be implemented to actually tell the component to turn on.

   Note that component *control* and component *monitoring* are decoupled from each other. So, for example, a component manager's `on()` method should not directly call the callback that tells the device that the component is now on. Rather, the command should return without calling the callback, and leave it to the *monitoring* to detect when the component has changed states.

   Consider, for example, a component that takes ten seconds to power up:

   1. The `on()` command should be implemented to tell the component to power up. If the component accepts this command without complaint, then the `on()` command should return success. The component manager should not, however, assume that the component is now on.

   2. After ten seconds, the component has powered up, and the component manager's monitoring detects that the component is on. Only then should the callback be called to let the device know that the component has changed state, resulting in a change of device state to `ON`.

Note: A component manager may maintain additional state, and support additional commands, that do not map to its component. That is, a call to a component manager needs not always result in a call to the underlying component. For example, a subarray's component manager may implement its `assign_resources` method simply to maintain a record (within the component manager itself) of what resources it has, so that it can validate arguments to other methods (for example, check that arguments to its `configure` method do not require access to resources that have not been assigned to it). In this case, the call to the component manager's `assign_resources` method would not result in interaction with the component; indeed, the component may not even possess the concepts of *resources* and *resource assignment*.

Implement command class objects

Tango device command functionality is implemented in command *classes* rather than methods. This allows for:

- functionality common to many classes to be abstracted out and implemented once for all. For example, there are many commands associated with transitional states (*e.g.* `Configure()` command and `CONFIGURING` state, `Scan()` command and `SCANNING` state, *etc.*). Command classes allow us to implement this association once for all, and to protect that implementation from accidental overriding by command subclasses.
- testing of commands independently of Tango. For example, a Tango device's `On()` command might only need to interact with the device's component manager and its operational state model. As such, in order to test the correct implementation of that command, we only need a component manager and an operational state model. Thus, we can test the command without actually instantiating the Tango device.

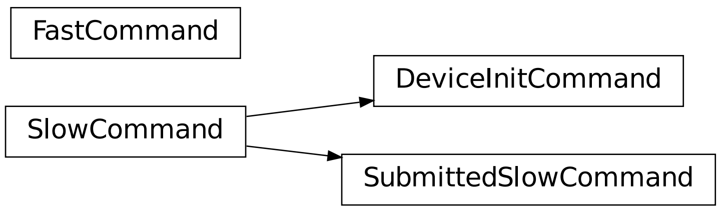
Writing a command class involves the following steps.

1. **Do you really need to implement the command?** If the command to be implemented is part of the Tango device you will inherit from, perhaps the current implementation is exactly what you need.

   For example, the `SKABaseDevice` class's implementation of the `On()` command simply calls its component manager's `on()` method. Maybe you don't need to change that; you've implemented your component manager's `on()` method, and that's all there is to do.

2. **Choose a command class to subclass.**

   - If the command to be implemented is part of the device you will inherit from (but you still need to override it), then you would generally subclass the base device's command class. For example, if if you need to override `SKABaseDevice`'s `Standby` command, then you would subclass `SKABaseDevice.StandbyCommand`.
   - If the command is a new command, not present in the base device class, then you will want to inherit from one or more command classes in the *`ska_tango_base.commands`* module.

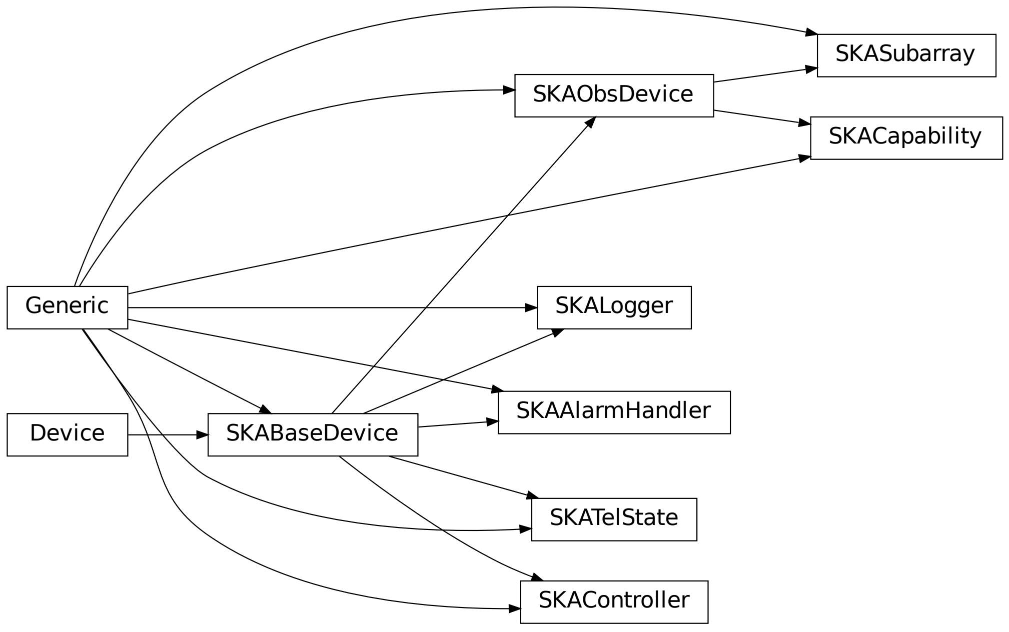


3. **Implement class methods.**
   - In many cases, you only need to implement the `do()` method.
   - To constrain when the command is allowed to be invoked, override the `is_allowed()` method.

Write your Tango device

Writing the Tango device involves the following steps:

1. **Select a device class to subclass.**



2. **Register your component manager.** This is done by overriding the `create_component_manager` class to return your component manager object:

```
def create_component_manager(self):
    return AntennaComponentManager(
        self.op_state_model, logger=self.logger
    )
```

3. **Implement commands.** You've already written the command classes. There is some boilerplate to ensure that the Tango command methods invoke the command classes:

   1. Registration occurs in the `init_command_objects` method, using calls to the `register_command_object` helper method. Implement the `init_command_objects` method:

   ```
   def init_command_objects(self):
       super().init_command_objects()

       self.register_command_object(
           "DoStuff", self.DoStuffCommand(self.component_manager, self.logger)
       )
       self.register_command_object(
           "DoOtherStuff", self.DoOtherStuffCommand(
               self.component_manager, self.logger
           )
       )
   ```

   2. Any new commands need to be implemented as:

   ```
   @command(dtype_in=..., dtype_out=...)
   def DoStuff(self, argin):
       command = self.get_command_object("DoStuff")
       return command(argin)
   ```

   or, if the command does not take an argument:

   ```
   @command(dtype_out=...)
   def DoStuff(self):
       command = self.get_command_object("DoStuff")
       return command()
   ```

   Note that these two examples deliberately push all SKA business logic down to the command class (at least) or even the component manager. It is highly recommended not to include SKA business logic in Tango devices. However, Tango-specific functionality can and should be implemented directly into the command method. For example, many SKA commands accept a JSON string as argument, as a workaround for the fact that Tango commands cannot accept more than one argument. Since this use of JSON is closely associated with Tango, we might choose to unpack our JSON strings in the command method itself, thus leaving our command objects free of JSON:

   ```
   @command(dtype_in=..., dtype_out=...)
   def DoStuff(self, argin):
       args = json.loads(argin)
       command = self.get_command_object("DoStuff")
       return command(args)
   ```

1.2 Components and component managers

A fundamental assumption of this package is that each Tango device exists to provide monitoring and control of some *component* of a SKA telescope.

A *component* could be (for example):

- Hardware such as an antenna, dish, atomic clock, TPM, switch, etc
- An external service such as a database or cluster workload manager
- A software process or thread launched by the Tango device.
- In a hierarchical system, a group of subservient Tango devices.

By analogy, if the *component* is a television, the Tango device would be the remote control for that television.

1.2.1 Tango devices and their components

Note the distinction between a component and the Tango device that is responsible for monitoring and controlling that component.

A component might be hardware equipment installed on site, such as a dish or an antenna. The Tango device that monitors that component is a software object, in a process running on a server, probably located in a server room some distance away. Thus the Tango device and its component are largely independent of each other:

- A Tango device may be running normally when its component is in a fault state, or turned off, or even not fitted. Device states like `OFF` and `FAULT` represent the state of the monitored component. A Tango device that reports `OFF` state is running normally, and reporting that its component is turned off. A Tango device that reports `FAULT` state is running normally, and reporting that its component is in a fault state.
- When a Tango device itself experiences a fault (for example, its server crashes), this is not expected to affect the component. The component continues to run; the only impact is it can no longer be monitored or controlled.

  By analogy: when the batteries in your TV remote control go flat, the TV continues to run.
- We should not assume that a component's state is governed solely by its Tango device. On the contrary, components are influenced by a wide range of factors. For example, the following are ways in which a component might be switched off:
  - Its Tango device switches it off via its software interface;
  - Some other software entity switches it off via its software interface;
  - The hardware switches itself off, or its firmware switches it off, because it detected a critical fault.
  - The equipment's power button is pressed;
  - An upstream power supply device denies it power.

  A Tango device therefore must not treat its component as under its sole control. For example, having turned its component on, it must not assume that the component will remain on. Rather, it must continually *monitor* its component, and update its state to reflect changes in component state.

1.2.2 Component monitoring

Component *monitoring* is the main mechanism by which a Tango device maintains and updates its state:

- A Tango device should not make assumptions about component state after issuing a command. For example, after successfully telling its component to turn on, a Tango device should not assume that the component is on, and transition immediately to ON state. Rather, it should wait for its monitoring of the component to provide confirmation that the component is on; only then should it transition to ON state. It follows that a Tango device's `On()` command might complete successfully, yet the device's `state()` might not report `ON` state immediately, or for some seconds, or indeed at all.
- A Tango device also should not make assumptions about component state when the Tango device is *initialising*. For example, in a normal controlled startup of a telescope, an initialising Tango device might expect to find its component switched off, and to be itself responsible for switching the component on at the proper time. However, this is not the only circumstance in which a Tango device might initialise; the Tango device would also have to initialise following a reboot of the server on which it runs. In such a case, the component might already be switched on. Thus, at initialisation, a Tango device should merely launch the component monitoring that will allows the device to detect the state of the component.

1.2.3 Component managers

A Tango device's responsibility to monitor and control its component is largely separate from its interface to the Tango subsystem. Therefore, devices in this package implement component monitoring and control in a separate *component manager*.

A component manager is responsible for:

- establishing and maintaining communication with the component. For example:
  - If the component interface is via a connection-oriented protocol (such as TCP/IP), then the component manager must establish and maintain a *connection* to the component;
  - If the component is able to publish updates, then the component manager would need to subscribe to those updates;
  - If the component cannot publish updates, but can only respond to requests, then the component manager would need to initiate polling of the component.
- implementing monitoring of the component so that changes in component state trigger callbacks that report those changes up to the Tango device;
- implementing commands such as `off()`, `on()`, etc., so that they actually tell the component to turn off, turn on, etc.

Note: It is highly recommended to implement your component manager, and thoroughly test it, *before* embedding it in a Tango device.

1.3 Long Running Commands

Many SKA device commands involve actions whose duration is inherently slow or unpredictable. For example, a command might need to interact with hardware, other devices, or other external systems over a network; read to or write from a file system; or perform intensive computation. If a TANGO device blocks while such a command runs, then there is a period of time in which it cannot respond to other requests. Its overall performance declines, and timeouts may even occur.

To address this, the base device provides long running commands (LRC) support, in the form of an interface and mechanism for running such commands asynchronously.

Note: Long Running Command: A TANGO command for which the execution time is in the order of seconds (CS Guidelines recommends less than 10 ms). In this context it also means a command which is implemented to execute asynchronously. Long running, slow command and asynchronous command are used interchangeably in this text and the code base. In the event where the meaning differ it will be explained but all refer to non-blocking calls.

This means that devices return immediately with a response while busy with the actual task in the background or parked on a queue pending the next available worker.

New attributes and commands have been added to the base device to support the mechanism to execute long running TANGO commands asynchronously.

1.3.1 Monitoring Progress of Long Running Commands

In addition to the listed requirements above, the device should provide monitoring points to allow clients determine when a LRC is received, executing or completed (success or fail). LRCs can assume any of the following defined task states: STAGING, QUEUED, IN_PROGRESS, ABORTED, NOT_FOUND, COMPLETED, REJECTED, FAILED.

A new set of attributes and commands have been added to the base device to enable monitoring and reporting of result, status and progress of LRCs.

LRC Attributes

Attribute	Example Value	Description
longRunningCommandsInQueue	('StandbyCommand', 'OnCommand', 'OffCommand')	Keeps track of which commands are on the queue
longRunningCommandIDsInQueue	('1636437568.0723004_235210334802782_OnCommand', 1636437789.493874_116219429722764_OffCommand)	Keeps track of IDs in the queue
longRunningCommandStatus	('1636437568.0723004_235210334802782_OnCommand', 'IN_PROGRESS', '1636437789.493874_116219429722764_OffCommand', 'IN_PROGRESS')	ID, status pair of the currently executing commands
longRunningCommandProgress	('1636437568.0723004_235210334802782_OnCommand', '12', '1636437789.493874_116219429722764_OffCommand', '1')	ID, progress pair of the currently executing commands
longRunningCommandResult	('1636438076.6105473_101143779281769_OnCommand', '0', 'OK')	ID, ResultCode, result of the completed command

LRC Commands

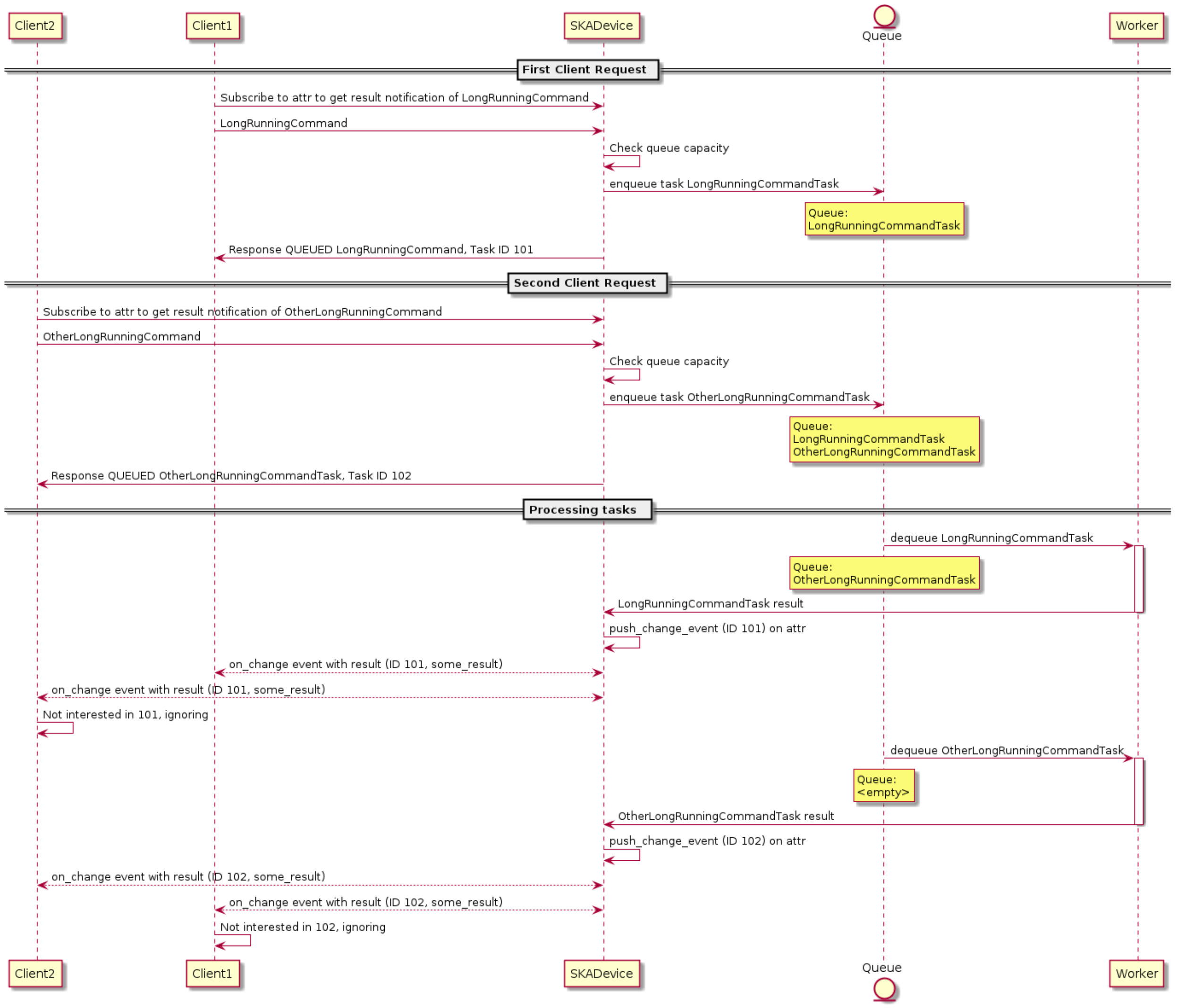
Command	Description
CheckLongRunningCommandStatus	Check the status of a long running command by ID
AbortCommands	Abort the currently executing LRCs and remove all enqueued LRCs

In addition to the set of commands in the table above, a number of candidate SKA commands in the base device previously implemented as blocking commands have been converted to execute as long running commands (asynchronously), viz: Standby, On, Off, Reset and GetVersionInfo.

The device has change events configured for all the LRC attributes which clients can use to track their requests. **The client has the responsibility of subscribing to events to receive changes on command status and results**.

1.3.2 UML Illustration

Multiple Clients Invoke Multiple Long Running Commands


Client2
Client1
SKADevice
Queue
Worker
First Client Request
Subscribe to attr to get result notification of LongRunningCommand
LongRunningCommand
Check queue capacity
enqueue task LongRunningCommandTask
Queue:
LongRunningCommandTask
Response QUEUED LongRunningCommand, Task ID 101
Second Client Request
Subscribe to attr to get result notification of OtherLongRunningCommand
OtherLongRunningCommand
Check queue capacity
enqueue task OtherLongRunningCommandTask
Queue:
LongRunningCommandTask
OtherLongRunningCommandTask
Response QUEUED OtherLongRunningCommandTask, Task ID 102
Processing tasks
dequeue LongRunningCommandTask
Queue:
OtherLongRunningCommandTask
LongRunningCommandTask result
push_change_event (ID 101) on attr
on_change event with result (ID 101, some_result)
on_change event with result (ID 101, some_result)
Not interested in 101, ignoring
dequeue OtherLongRunningCommandTask
Queue:
<empty>
OtherLongRunningCommandTask result
push_change_event (ID 102) on attr
on_change event with result (ID 102, some_result)
on_change event with result (ID 102, some_result)
Not interested in 102, ignoring
Client2
Client1
SKADevice
Queue
Worker

1.3.3 How to implement a long running command using the provided executor

A task executor has been provisioned to handle the asynchronous execution of tasks put on the queue. Your sample component manager will be asynchronous if it inherits from the provisioned executor. You can also swap out the default executor with any asynchronous mechanism for your component manager.

Create a component manager

```python
class SampleComponentManager(TaskExecutorComponentManager):
    """A sample component manager"""

    def __init__(
        self,
        *args,
        max_workers: Optional[int] = None,
        logger: logging.Logger = None,
        **kwargs,
    ):
        """Init SampleComponentManager."""

        # Set up your class

        super().__init__(*args, max_workers=max_workers, logger=logger, **kwargs)
```

Add a method that should be executed in a background thread

```python
# class SampleComponentManager

    def _a_very_slow_method(
        logger: logging.Logger,
        task_callback: Callable,
        task_abort_event: Event,
    ):
        """This is a long running method

        :param logger: logger
        :param task_callback: Update task state, defaults to None
        :param task_abort_event: Check for abort, defaults to None
        """
        # Indicate that the task has started
        task_callback(status=TaskStatus.IN_PROGRESS)
        for current_iteration in range(100):
            # Update the task progress
            task_callback(progress=current_iteration)

            # Do something
            time.sleep(10)

            # Periodically check that tasks have not been ABORTED
            if task_abort_event.is_set():
                # Indicate that the task has been aborted
```

(continues on next page)

(continued from previous page)

```
                task_callback(status=TaskStatus.ABORTED, result="This task aborted")
                return

        # Indicate that the task has completed
        task_callback(status=TaskStatus.COMPLETED, result="This slow task has completed")
```

Add a method to submit the slow method

```
# class SampleComponentManager

    def submit_slow_method(self, task_callback: Optional[Callable] = None):
        """Submit the slow task.

        This method returns immediately after it submitted
        `self._a_very_slow_method` for execution.

        :param task_callback: Update task state, defaults to None
        """
        task_status, response = self.submit_task(
            self._a_very_slow_method, args=[], task_callback=task_callback
        )
        return task_status, response
```

Create the component manager in your Tango device

```
class SampleDevice(SKABaseDevice):
    """A sample Tango device"""

    def create_component_manager(self):
        """Create a component manager."""
        return SampleComponentManager(
            max_workers=2,
            logger=self.logger,
            communication_state_callback=self._communication_state_changed,
            component_state_callback=self._component_state_changed,
        )
```

Init the command object

```
# class SampleDevice(SKABaseDevice):

    def init_command_objects(self):
        """Initialise the command handlers."""
        super().init_command_objects()

        ...

```

(continues on next page)

(continued from previous page)

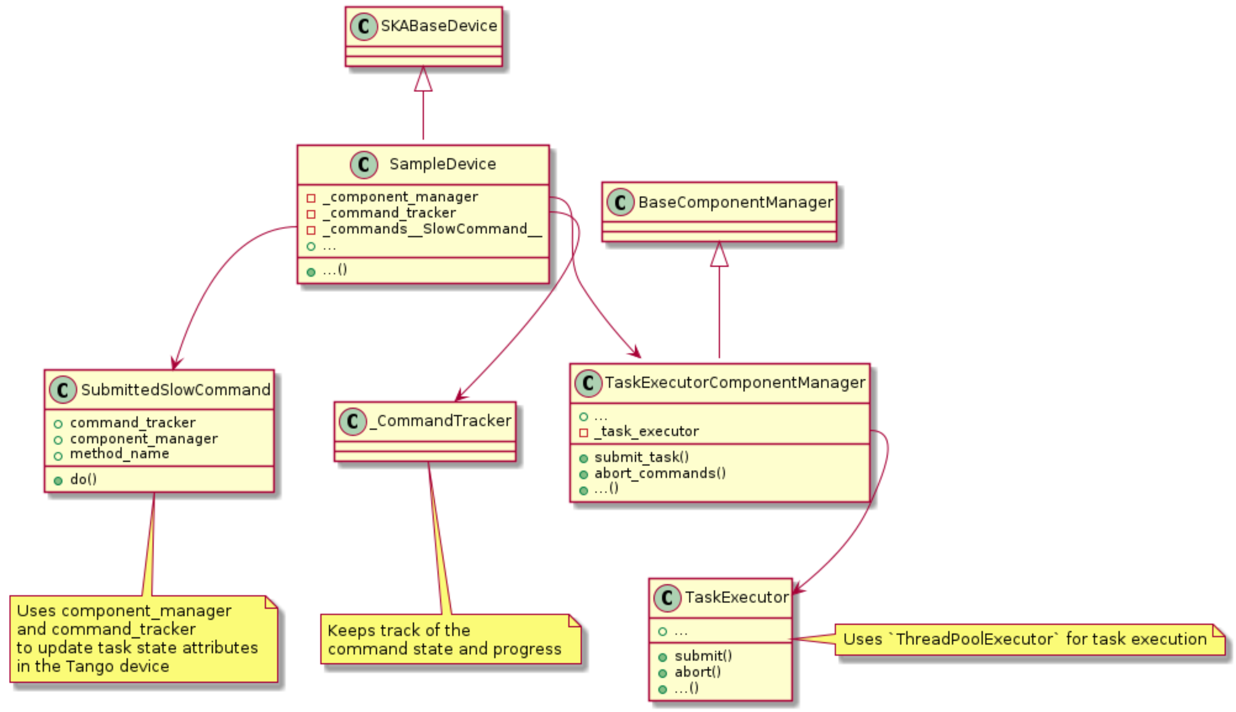
```
        self.register_command_object(
            "VerySlow",
            SubmittedSlowCommand(
                "VerySlow",
                self._command_tracker,
                self.component_manager,
                "submit_slow_method",
                callback=None,
                logger=self.logger,
            ),
        )
```

Create the Tango Command

```
# class SampleDevice(SKABaseDevice):

    @command(
        dtype_in=None,
        dtype_out="DevVarStringArray",
    )
    @DebugIt()
    def VerySlow(self):
        """A very slow command."""
        handler = self.get_command_object("VerySlow")
        (return_code, message) = handler()
        return f"{return_code}", message
```

1.3.4 Class diagram



CHAPTER

TWO

API

2.1 Base subpackage

This subpackage implements functionality common to all SKA Tango devices.

2.1.1 Admin Mode Model

This module provides the admin mode model for SKA LMC Tango devices.

The model is now defined in the `ska_control_model` package, but is imported here for backwards compatibility.

2.1.2 Op State Model

This module provides the op state model for SKA LMC Tango devices.

The model is now defined in the `ska_control_model` package, but is imported here for backwards compatibility.

2.1.3 Logging

This module implements the logging framework for the SKA base device.

2.1.4 Base Component Manager

This module provides an abstract component manager for SKA Tango base devices.

The basic model is:

- Every Tango device has a *component* that it monitors and/or controls. That component could be, for example:
  - Hardware such as an antenna, APIU, TPM, switch, subrack, etc.
  - An external software system such as a cluster manager
  - A software routine, possibly implemented within the Tango device itself
  - In a hierarchical system, a pool of lower-level Tango devices.
- A Tango device will usually need to establish and maintain *communication* with its component. This connection may be deliberately broken by the device, or it may fail.
- A Tango device *controls* its component by issuing commands that cause the component to change behaviour and/or state; and it *monitors* its component by keeping track of its state.

check_communicating(*func: Wrapped*) → Wrapped

Return a function that checks component communication before calling a function.

The component manager needs to have established communications with the component, in order for the function to be called.

This function is intended to be used as a decorator:

```
@check_communicating
def scan(self):
    ...
```

Parameters
`func` – the wrapped function

Returns
the wrapped function

check_on(*func: Wrapped*) → Wrapped

Return a function that checks the component state then calls another function.

The component needs to be turned on, and not faulty, in order for the function to be called.

This function is intended to be used as a decorator:

```
@check_on
def scan(self):
    ...
```

Parameters
`func` – the wrapped function

Returns
the wrapped function

class BaseComponentManager(*logger: Logger*, *communication_state_callback: Optional[Callable[[CommunicationStatus], None]] = None*, *component_state_callback: Optional[Callable[[...], None]] = None*, ***state: Any*)

An abstract base class for a component manager for SKA Tango devices.

It supports:

- Maintaining a connection to its component
- Controlling its component via commands like Off(), Standby(), On(), etc.
- Monitoring its component, e.g. detect that it has been turned off or on

start_communicating() → None

Establish communication with the component, then start monitoring.

This is the place to do things like:

- Initiate a connection to the component (if your communication is connection-oriented)
- Subscribe to component events (if using "pull" model)
- Start a polling loop to monitor the component (if using a "push" model)

Raises
`NotImplementedError` – Not implemented it's an abstract class

`stop_communicating()` → None

Cease monitoring the component, and break off all communication with it.

For example,

- If you are communicating over a connection, disconnect.
- If you have subscribed to events, unsubscribe.
- If you are running a polling loop, stop it.

Raises
`NotImplementedError` – Not implemented it's an abstract class

`property communication_state: CommunicationStatus`

Return the communication status of this component manager.

Returns
status of the communication channel with the component.

`property component_state: dict[str, Any]`

Return the state of this component manager's component.

Returns
state of the component.

`off`(*task_callback: Optional[Callable] = None*) → tuple[*TaskStatus*, str]

Turn the component off.

Parameters
`task_callback` – callback to be called when the status of the command changes

Raises
`NotImplementedError` – Not implemented it's an abstract class

`standby`(*task_callback: Optional[Callable] = None*) → tuple[*TaskStatus*, str]

Put the component into low-power standby mode.

Parameters
`task_callback` – callback to be called when the status of the command changes

Raises
`NotImplementedError` – Not implemented it's an abstract class

`on`(*task_callback: Optional[Callable] = None*) → tuple[*TaskStatus*, str]

Turn the component on.

Parameters
`task_callback` – callback to be called when the status of the command changes

Raises
`NotImplementedError` – Not implemented it's an abstract class

`reset`(*task_callback: Optional[Callable] = None*) → tuple[*TaskStatus*, str]

Reset the component (from fault state).

Parameters
`task_callback` – callback to be called when the status of the command changes

Raises
`NotImplementedError` – Not implemented it's an abstract class

`abort_commands`(*task_callback: Optional[Callable] = None*) → tuple[*TaskStatus*, str]

Abort all tasks queued & running.

Parameters
`task_callback` – callback to be called whenever the status of the task changes.

Raises
`NotImplementedError` – Not implemented it's an abstract class

2.1.5 Base Device

This module implements a generic base model and device for SKA.

It exposes the generic attributes, properties and commands of an SKA device.

class `SKABaseDevice`(**args: Any*, ***kwargs: Any*)

A generic base device for SKA.

class `InitCommand`(*device: tango.server.Device*, *logger: Optional[Logger] = None*)

A class for the SKABaseDevice's init_device() "command".

`do`(**args: Any*, ***kwargs: Any*) → tuple[*ResultCode*, str]

Stateless hook for device initialisation.

Parameters
- `args` – positional arguments to this do method
- `kwargs` – keyword arguments to this do method

Returns
A tuple containing a return code and a string message indicating status. The message is for information purpose only.

`SkaLevel`

Device property.

Indication of importance of the device in the SKA hierarchy to support drill-down navigation: 1..6, with 1 highest.

`GroupDefinitions`

Device property.

Each string in the list is a JSON serialised dict defining the `group_name`, `devices` and `subgroups` in the group. A Tango Group object is created for each item in the list, according to the hierarchy defined. This provides easy access to the managed devices in bulk, or individually.

The general format of the list is as follows, with optional `devices` and `subgroups` keys:

```
[ {"group_name": "<name>",
   "devices": ["<dev name>", ...]},
  {"group_name": "<name>",
   "devices": ["<dev name>", "<dev name>", ...],
   "subgroups" : [{<nested group>},
                  {<nested group>}, ...]},
  ...
  ]
```

For example, a hierarchy of racks, servers and switches:

```
[ {"group_name": "servers",
   "devices": ["elt/server/1", "elt/server/2",
                "elt/server/3", "elt/server/4"]},
  {"group_name": "switches",
   "devices": ["elt/switch/A", "elt/switch/B"]},
  {"group_name": "pdus",
   "devices": ["elt/pdu/rackA", "elt/pdu/rackB"]},
  {"group_name": "racks",
   "subgroups": [
        {"group_name": "rackA",
         "devices": ["elt/server/1", "elt/server/2",
                      "elt/switch/A", "elt/pdu/rackA"]},
        {"group_name": "rackB",
         "devices": ["elt/server/3", "elt/server/4",
                      "elt/switch/B", "elt/pdu/rackB"],
         "subgroups": []}
   ]} ]
```

LoggingLevelDefault

Device property.

Default logging level at device startup. See `LoggingLevel`

LoggingTargetsDefault

Device property.

Default logging targets at device startup. See the project readme for details.

init_device() → None

Initialise the tango device after startup.

Subclasses that have no need to override the default implementation of state management may leave `init_device()` alone. Override the `do()` method on the nested class `InitCommand` instead.

set_logging_level(*value: LoggingLevel*) → None

Set the logging level for the device.

Both the Python logger and the Tango logger are updated.

Parameters
value – Logging level for logger

Raises
LoggingLevelError – for invalid value

set_logging_targets(*targets: list[str]*) → None

Set the additional logging targets for the device.

Note that this excludes the handlers provided by the ska_ser_logging library defaults.

Parameters
targets – Logging targets for logger

create_component_manager() → ComponentManagerT

Create and return a component manager for this device.

Raises
NotImplementedError – for no implementation

`register_command_object`(*command_name: str*, *command_object:* FastCommand | SlowCommand) → None

Register an object as a handler for a command.

Parameters

- **`command_name`** – name of the command for which the object is being registered
- **`command_object`** – the object that will handle invocations of the given command

`get_command_object`(*command_name: str*) → *FastCommand* | *SlowCommand*

Return the command object (handler) for a given command.

Parameters
`command_name` – name of the command for which a command object (handler) is sought

Returns
the registered command object (handler) for the command

`init_command_objects`() → None

Register command objects (handlers) for this device's commands.

`buildState`() → str

Read the Build State of the device.

Returns
the build state of the device

`versionId`() → str

Read the Version Id of the device.

Returns
the version id of the device

`loggingLevel`(*value: LoggingLevel*) → None

Set the logging level for the device.

Both the Python logger and the Tango logger are updated.

Parameters
`value` – Logging level for logger

`loggingTargets`(*value: list[str]*) → None

Set the additional logging targets for the device.

Note that this excludes the handlers provided by the ska_ser_logging library defaults.

Parameters
`value` – Logging targets for logger

`healthState`() → HealthState

Read the Health State of the device.

It interprets the current device condition and condition of all managed devices to set this. Most possibly an aggregate attribute.

Returns
Health State of the device

`adminMode`(*value: AdminMode*) → None

Set the Admin Mode of the device.

Parameters
`value` – Admin Mode of the device.

Raises
`ValueError` – for unknown adminMode

`controlMode`(*value: ControlMode*) → None

Set the Control Mode of the device.

Parameters
`value` – Control mode value

`simulationMode`(*value: SimulationMode*) → None

Set the Simulation Mode of the device.

Parameters
`value` – SimulationMode

`testMode`(*value: TestMode*) → None

Set the Test Mode of the device.

Parameters
`value` – Test Mode

`longRunningCommandsInQueue`() → list[str]

Read the long running commands in the queue.

Keep track of which commands are in the queue. Pop off from front as they complete.

Returns
tasks in the queue

`longRunningCommandIDsInQueue`() → list[str]

Read the IDs of the long running commands in the queue.

Every client that executes a command will receive a command ID as response. Keep track of IDs in the queue. Pop off from front as they complete.

Returns
unique ids for the enqueued commands

`longRunningCommandStatus`() → list[str]

Read the status of the currently executing long running commands.

ID, status pair of the currently executing command. Clients can subscribe to on_change event and wait for the ID they are interested in.

Returns
ID, status pairs of the currently executing commands

`longRunningCommandProgress`() → list[str]

Read the progress of the currently executing long running command.

ID, progress of the currently executing command. Clients can subscribe to on_change event and wait for the ID they are interested in.

Returns
ID, progress of the currently executing command.

`longRunningCommandResult()` → tuple[str, str]

Read the result of the completed long running command.

Reports unique_id, json-encoded result. Clients can subscribe to on_change event and wait for the ID they are interested in.

Returns
ID, result.

`is_Reset_allowed()` → bool

Return whether the *Reset* command may be called in the current device state.

Returns
whether the command may be called in the current device state

`is_Standby_allowed()` → bool

Return whether the *Standby* command may be called in the current device state.

Returns
whether the command may be called in the current device state

`is_Off_allowed()` → bool

Return whether the *Off* command may be called in the current device state.

Returns
whether the command may be called in the current device state

`is_On_allowed()` → bool

Return whether the *On* command may be called in the current device state.

Returns
whether the command may be called in the current device state

`class AbortCommandsCommand`(*component_manager: ComponentManagerT*, *logger: Optional[Logger] = None*)

The command class for the AbortCommand command.

`do`(**args: Any*, ***kwargs: Any*) → tuple[*ResultCode*, str]

Abort long running commands.

Abort the currently executing LRC and remove all enqueued LRCs.

Parameters
- **args** – positional arguments to this do method
- **kwargs** – keyword arguments to this do method

Returns
A tuple containing a return code and a string message indicating status. The message is for information purpose only.

`class CheckLongRunningCommandStatusCommand`(*command_tracker:* CommandTracker, *logger: Optional[Logger] = None*)

The command class for the CheckLongRunningCommandStatus command.

`do`(**args: Any*, ***kwargs: Any*) → str

Determine the status of the command ID passed in, if any.
- Check *command_result* to see if it's finished.
- Check *command_status* to see if it's in progress
- Check *command_ids_in_queue* to see if it's queued

Parameters

- **args** – positional arguments to this do method. There should be only one: the command_id.
- **kwargs** – keyword arguments to this do method

Returns
The string of the TaskStatus

`class DebugDeviceCommand`(*device: tango.server.Device*, *logger: Optional[Logger] = None*)

A class for the SKABaseDevice's DebugDevice() command.

`do`(**args: Any*, ***kwargs: Any*) → int

Stateless hook for device DebugDevice() command.

Starts the `debugpy` debugger listening for remote connections (via Debugger Adaptor Protocol), and patches all methods so that they can be debugged.

If the debugger is already listening, additional execution of this command will trigger a breakpoint.

Parameters
- **args** – positional arguments to this do method
- **kwargs** – keyword arguments to this do method

Returns
The TCP port the debugger is listening on.

`start_debugger_and_get_port`(*port: int*) → int

Start the debugger and return the allocated port.

Parameters
port – port to listen on

Returns
allocated port

`monkey_patch_all_methods_for_debugger`() → None

Monkeypatch methods that need to be patched for the debugger.

`get_all_methods`() → list[tuple[object, str, Any]]

Return a list of the device's methods.

Returns
list of device methods

`static method_must_be_patched_for_debugger`(*owner: object*, *method: method*) → bool

Determine if methods are worth debugging.

The goal is to find all the user's Python methods, but not the lower level PyTango device and Boost extension methods. The *types.FunctionType* check excludes the Boost methods.

Parameters
- **owner** – owner
- **method** – the name

Returns
True if the method contains more than the skipped modules.

`patch_method_for_debugger`(*owner: object*, *name: str*, *method: object*) → None

Ensure method calls trigger the debugger.

Most methods in a device are executed by calls from threads spawned by the cppTango layer. These threads are not known to Python, so we have to explicitly inform the debugger about them.

Parameters
- **owner** – owner
- **name** – the name
- **method** – method

`set_state`(*state: tango.DevState*) → None

Set the device server state.

This is dependent on whether the set state call has been actioned from a native python thread or a tango omni thread

Parameters
`state` – the new device state

`set_status`(*status: str*) → None

Set the device server status string.

This is dependent on whether the set status call has been actioned from a native python thread or a tango omni thread

Parameters
`status` – the new device status

`push_change_event`(*name: str*, *value: Optional[Any] = None*) → None

Push a device server change event.

This is dependent on whether the push_change_event call has been actioned from a native python thread or a tango omni thread

Parameters

- `name` – the event name
- `value` – the event value

`push_archive_event`(*name: str*, *value: Optional[Any] = None*) → None

Push a device server archive event.

This is dependent on whether the push_archive_event call has been actioned from a native python thread or a tango omnithread.

Parameters

- `name` – the event name
- `value` – the event value

`add_attribute`(**args: Any*, ***kwargs: Any*) → None

Add a device attribute.

This is dependent on whether the push_archive_event call has been actioned from a native python thread or a tango omni thread

Parameters

- `args` – positional args
- `kwargs` – keyword args

`set_change_event`(*name: str*, *implemented: bool*, *detect: bool = True*) → None

Set an attribute's change event.

This is dependent on whether the push_archive_event call has been actioned from a native python thread or a tango omni thread

Parameters

- `name` – name of the attribute
- `implemented` – whether the device pushes change events

- **detect** – whether the Tango layer should verify the change event property

ExecutePendingOperations() → None

Execute any Tango operations that have been pushed on the queue.

The poll time is initially 5ms, to circumvent the problem of device initialisation, but is reset to 100ms after the first pass.

main(**args: str*, ***kwargs: str*) → int

Entry point for module.

Parameters

- **args** – positional arguments
- **kwargs** – named arguments

Returns
exit code

class CommandTracker(*queue_changed_callback: Callable[[list[tuple[str, str]]], None]*, *status_changed_callback: Callable[[list[tuple[str, TaskStatus]]], None]*, *progress_changed_callback: Callable[[list[tuple[str, int]]], None]*, *result_callback: Callable[[str, tuple[ResultCode, str]], None]*, *exception_callback: Optional[Callable[[str, Exception], None]] = None*, *removal_time: float = 10.0*)

A class for keeping track of the state and progress of commands.

new_command(*command_name: str*, *completed_callback: Optional[Callable[[], None]] = None*) → str

Create a new command.

Parameters

- **command_name** – the command name
- **completed_callback** – an optional callback for command completion

Returns
a unique command id

update_command_info(*command_id: str*, *status: Optional[TaskStatus] = None*, *progress: Optional[int] = None*, *result: Optional[tuple[ResultCode, str]] = None*, *exception: Optional[Exception] = None*) → None

Update status information on the command.

Parameters

- **command_id** – the unique command id
- **status** – the status of the asynchronous task
- **progress** – the progress of the asynchronous task
- **result** – the result of the completed asynchronous task
- **exception** – any exception caught in the running task

property commands_in_queue: list[tuple[str, str]]

Return a list of commands in the queue.

Returns
a list of (command_id, command_name) tuples, ordered by when invoked.

property command_statuses: `list[tuple[str, TaskStatus]]`

Return a list of command statuses for commands in the queue.

Returns
a list of (command_id, status) tuples, ordered by when invoked.

property command_progresses: `list[tuple[str, int]]`

Return a list of command progresses for commands in the queue.

Returns
a list of (command_id, progress) tuples, ordered by when invoked.

property command_result: `Optional[tuple[str, Optional[tuple[ResultCode, str]]]]`

Return the result of the most recently completed command.

Returns
a (command_id, result) tuple. If no command has completed yet, then None.

property command_exception: `Optional[tuple[str, Exception]]`

Return the most recent exception, if any.

Returns
a (command_id, exception) tuple. If no command has raised an uncaught exception, then None.

get_command_status(*command_id: str*) → *TaskStatus*

Return the current status of a running command.

Parameters
command_id – the unique command id

Returns
a status of the asynchronous task.

2.2 Executor subpackage

This subpackage provides a generic and flexible polling mechanism.

2.2.1 Executor

This module provides for asynchronous execution of tasks.

class TaskExecutor(*max_workers: Optional[int]*)

An asynchronous executor of tasks.

submit(*func: Callable*, *args: Optional[Any] = None*, *kwargs: Optional[Any] = None*, *task_callback: Optional[Callable] = None*) → tuple[*TaskStatus*, str]

Submit a new task.

Parameters

- **func** – the function to be executed.
- **args** – positional arguments to the function
- **kwargs** – keyword arguments to the function

- `task_callback` – the callback to be called when the status or progress of the task execution changes

Returns
(TaskStatus, message)

`abort`(*task_callback: Optional[Callable] = None*) → tuple[*TaskStatus*, str]

Tell this executor to abort execution.

New submissions will be rejected until the queue is empty and no tasks are still running. Tasks on the queue will be marked as aborted and not run. Tasks already running will be allowed to continue running

Parameters
`task_callback` – callback for abort complete

Returns
tuple of task status & message

`class TaskStatus`(*value*)

The status of a task.

A task is any operation that is being performed asynchronously.

`STAGING = 0`

The request to execute the task has not yet been acted upon.

`QUEUED = 1`

The task has been accepted and will be executed at a future time.

`IN_PROGRESS = 2`

The task is being executed.

`ABORTED = 3`

The task has been aborted.

`NOT_FOUND = 4`

The task is not found.

`COMPLETED = 5`

The task was completed.

Note that this does not necessarily imply that the task was executed successfully. Whether the task succeeded or failed is a matter for the `ResultCode`. The `COMPLETED` value indicates only that execution of the task ran to completion.

`REJECTED = 6`

The task was rejected.

`FAILED = 7`

The task failed to complete.

Note that this should not be used for a task that executes to completion, but does not achieve its goal. This kind of domain-specific notion of “succeeded” versus “failed” should be passed in a `ResultCode`. Here, `FAILED` means that the task executor has detected a failure of the task to run to completion. For example, execution of the task might have resulted in the raising of an uncaught exception.

2.2.2 Executor component manager

This module provides an abstract component manager for SKA Tango base devices.

class TaskExecutorComponentManager(**args: Any*, *max_workers: Optional[int] = None*, ***kwargs: Any*)

A component manager with support for asynchronous tasking.

submit_task(*func: Callable*, *args: Optional[Any] = None*, *kwargs: Optional[Any] = None*, *task_callback: Optional[Callable] = None*) → tuple[*TaskStatus*, str]

Submit a task to the task executor.

Parameters

- **func** – function/bound method to be run
- **args** – positional arguments to the function
- **kwargs** – keyword arguments to the function
- **task_callback** – callback to be called whenever the status of the task changes.

Returns
tuple of taskstatus & message

abort_commands(*task_callback: Optional[Callable] = None*) → tuple[*TaskStatus*, str]

Tell the task executor to abort all tasks.

Parameters
task_callback – callback to be called whenever the status of this abort task changes.

Returns
tuple of taskstatus & message

2.3 Obs subpackage

This subpackage models an SKA Tango observing device.

2.3.1 Obs State Model

This module provides the state model for observation state.

The model is now defined in the `ska_control_model` package, but is imported here for backwards compatibility.

2.3.2 Obs Device

SKAObsDevice.

A generic base device for Observations for SKA. It inherits SKABaseDevice class. Any device implementing an obsMode will inherit from SKAObsDevice instead of just SKABaseDevice.

class ObsDeviceComponentManager(*logger: Logger*, *communication_state_callback: Optional[Callable[[CommunicationStatus], None]] = None*, *component_state_callback: Optional[Callable[[...], None]] = None*, ***state: Any*)

A stub for an observing device component manager.

`class SKAObsDevice`(**args: Any*, ***kwargs: Any*)

A generic base device for Observations for SKA.

`class InitCommand`(*device: tango.server.Device*, *logger: Optional[Logger] = None*)

A class for the SKAObsDevice's init_device() "command".

`do`(**args: Any*, ***kwargs: Any*) → tuple[*ResultCode*, str]

Stateless hook for device initialisation.

Parameters

- **args** – positional arguments to the command. This command does not take any, so this should be empty.
- **kwargs** – keyword arguments to the command. This command does not take any, so this should be empty.

Returns

A tuple containing a return code and a string message indicating status. The message is for information purpose only.

`create_component_manager()` → ComponentManagerT

Create and return a component manager for this device.

Raises

`NotImplementedError` – because it is not implemented.

`obsState()` → ObsState

Read the Observation State of the device.

Returns

the current obs_state value

`obsMode()` → ObsMode

Read the Observation Mode of the device.

Returns

the current obs_mode value

`configurationProgress()` → int

Read the percentage configuration progress of the device.

Returns

the percentage configuration progress

`configurationDelayExpected()` → int

Read the expected Configuration Delay in seconds.

Returns

the expected configuration delay

`main`(**args: str*, ***kwargs: str*) → int

Entry point for module.

Parameters

- **args** – positional arguments
- **kwargs** – named arguments

Returns

exit code

2.4 Poller subpackage

This subpackage provides a generic and flexible polling mechanism.

2.4.1 Poller

This module provides a general framework and mechanism for polling.

`class Poller`(*poll_model:* PollModel, *poll_rate: float = 1.0*)

A generic hardware polling mechanism.

`start_polling`() → None

Start polling.

`stop_polling`() → None

Stop polling.

`class PollModel`(**args*, ***kwds*)

Abstract base class for a polling model.

`get_request`() → PollRequestT

Return the polling request to be executed at the next poll.

This is a hook called by the poller each polling loop, to obtain instructions on what it should do on the next poll.

Returns
attribute request to be executed at the next poll.

Raises
`NotImplementedError` – because this class is abstract

`poll`(*poll_request: PollRequestT*) → PollResponseT

Perform a single poll.

This is a hook called by the poller each polling loop.

Parameters
`poll_request` – specification of what is to be done on the poll. It might, for example, contain a list of reads and writes to be executed.

Returns
responses from this poll

Raises
`NotImplementedError` – because this class is abstract.

`polling_started`() → None

Respond to polling having started.

This is a hook called by the poller when it starts polling.

`polling_stopped`() → None

Respond to polling having stopped.

This is a hook called by the poller when it stops polling.

`poll_succeeded`(*poll_response: PollResponseT*) → None

Handle successful completion of a poll.

This is a hook called by the poller upon the successful completion of a poll.

Parameters

`poll_response` – The response to the poll, containing for example any values read.

`poll_failed`(*exception: Exception*) → None

Respond to an exception being raised by a poll attempt.

This is a hook called by the poller when an exception occurs. The polling loop itself never raises exceptions. It catches everything and simply calls this hook to let the polling model know what it caught.

Parameters

`exception` – the exception that was raised by a recent poll attempt.

`PollRequestT`

Type variable for object specifying what the poller should do next poll.

alias of TypeVar('PollRequestT')

`PollResponseT`

Type variable for object containing the result of the previous poll.

alias of TypeVar('PollResponseT')

2.4.2 Polling component manager

This module provides a polling component manager.

`class PollingComponentManager`(*logger: Logger*, *communication_state_callback: Callable*, *component_state_callback: Callable*, *poll_rate: float = 0.1*, ***kwargs: Any*)

Abstract base class for a component manager that polls its component.

`start_communicating`() → None

Start polling the component.

`polling_started`() → None

Respond to polling having started.

This is a hook called by the poller when it starts polling.

`stop_communicating`() → None

Stop polling the spectrum analyser.

`polling_stopped`() → None

Respond to polling having stopped.

This is a hook called by the poller when it stops polling.

`poll_failed`(*exception: Exception*) → None

Respond to an exception being raised by a poll attempt.

This is a hook called by the poller when an exception occurs.

Parameters

`exception` – the exception that was raised by a recent poll attempt.

`poll_succeeded`(*poll_response: PollResponseT*) → None

Handle a successful poll, including any values received.

This is a hook called by the poller at the end of each successful poll.

Parameters
`poll_response` – response to the poll, including any values received.

`get_request`() → PollRequestT

Return the reads and writes to be executed in the next poll.

Raises
`NotImplementedError` – because this class is abstract.

Returns
reads and writes to be executed in the next poll.

`poll`(*poll_request: PollRequestT*) → PollResponseT

Poll the hardware.

Connect to the hardware, write any values that are to be written, and then read all values.

Parameters
`poll_request` – specification of the reads and writes to be performed in this poll.

Raises
`NotImplementedError` – because this class is abstract.

Returns
responses to queries in this poll

2.5 Subarray

This subpackage models a SKA subarray Tango device.

2.5.1 Subarray Component Manager

This module provides an abstract component manager for SKA Tango subarray devices.

class `SubarrayComponentManager`(*logger: Logger, communication_state_callback: Optional[Callable[[CommunicationStatus], None]] = None, component_state_callback: Optional[Callable[[...], None]] = None, **state: Any*)

An abstract base class for a component manager for an SKA subarray Tango devices.

It supports:

- Maintaining a connection to its component
- Controlling its component via commands like AssignResources(), Configure(), Scan(), etc.
- Monitoring its component, e.g. detect that a scan has completed

`assign`(*resources: set[str], task_callback: Optional[Callable[[], None]] = None*) → tuple[*TaskStatus*, str]

Assign resources to the component.

Parameters

- `resources` – resources to be assigned

- **task_callback** – callback to be called when the status of the command changes

Raises
NotImplementedError – This is an abstract class

release(*resources: set[str], task_callback: Optional[Callable[[], None]] = None*) → tuple[*TaskStatus*, str]

Release resources from the component.

Parameters

- **resources** – resources to be released
- **task_callback** – callback to be called when the status of the command changes

Raises
NotImplementedError – This is an abstract class

release_all(*task_callback: Optional[Callable] = None*) → tuple[*TaskStatus*, str]

Release all resources.

Parameters
task_callback – callback to be called when the status of the command changes

Raises
NotImplementedError – This is an abstract class

configure(*configuration: dict[str, Any], task_callback: Optional[Callable[[], None]] = None*) → tuple[*TaskStatus*, str]

Configure the component.

Parameters

- **configuration** – the configuration to be configured
- **task_callback** – callback to be called when the status of the command changes

Raises
NotImplementedError – This is an abstract class

deconfigure(*task_callback: Optional[Callable] = None*) → tuple[*TaskStatus*, str]

Deconfigure this component.

Parameters
task_callback – callback to be called when the status of the command changes

Raises
NotImplementedError – This is an abstract class

scan(*args: str, task_callback: Optional[Callable] = None*) → tuple[*TaskStatus*, str]

Start scanning.

Parameters

- **args** – scan parameters encoded in a json string
- **task_callback** – callback to be called when the status of the command changes

Raises
NotImplementedError – This is an abstract class

end_scan(*task_callback: Optional[Callable] = None*) → tuple[*TaskStatus*, str]

End scanning.

Parameters
task_callback – callback to be called when the status of the command changes

Raises
`NotImplementedError` – This is an abstract class

`abort`(*task_callback: Optional[Callable] = None*) → tuple[*TaskStatus*, str]

Tell the component to abort whatever it was doing.

Parameters
`task_callback` – callback to be called when the status of the command changes

Raises
`NotImplementedError` – This is an abstract class

`obsreset`(*task_callback: Optional[Callable] = None*) → tuple[*TaskStatus*, str]

Reset the component to unconfigured but do not release resources.

Parameters
`task_callback` – callback to be called when the status of the command changes

Raises
`NotImplementedError` – This is an abstract class

`restart`(*task_callback: Optional[Callable] = None*) → tuple[*TaskStatus*, str]

Deconfigure and release all resources.

Parameters
`task_callback` – callback to be called when the status of the command changes

Raises
`NotImplementedError` – This is an abstract class

`property assigned_resources: list[str]`

Return the resources assigned to the component.

Raises
`NotImplementedError` – the resources assigned to the component

`property configured_capabilities: list[str]`

Return the configured capabilities of the component.

Raises
`NotImplementedError` – list of strings indicating number of configured instances of each capability type

2.5.2 Subarray Device

SKASubarray.

A SubArray handling device. It allows the assigning/releasing of resources into/from Subarray, configuring capabilities, and exposes the related information like assigned resources, configured capabilities, etc.

`class SKASubarray`(**args: Any, **kwargs: Any*)

Implements the SKA SubArray device.

`class InitCommand`(*device: tango.server.Device, logger: Optional[Logger] = None*)

A class for the SKASubarray's init_device() "command".

`do`(**args: Any, **kwargs: Any*) → tuple[*ResultCode*, str]

Stateless hook for device initialisation.

Parameters

- **args** – positional arguments to the command. This command does not take any, so this should be empty.
- **kwargs** – keyword arguments to the command. This command does not take any, so this should be empty.

Returns
A tuple containing a return code and a string message indicating status. The message is for information purpose only.

`class AbortCommand`(*command_tracker:* CommandTracker, *component_manager:* SubarrayComponentManager, *callback: Callable[[], None]*, *logger: Optional[Logger] = None*)

A class for SKASubarray's Abort() command.

`do`(**args: Any*, ***kwargs: Any*) → tuple[*ResultCode*, str]

Stateless hook for Abort() command functionality.

Parameters
- **args** – positional arguments to the command. This command does not take any, so this should be empty.
- **kwargs** – keyword arguments to the command. This command does not take any, so this should be empty.

Returns
A tuple containing a return code and a string message indicating status. The message is for information purpose only.

`create_component_manager()` → ComponentManagerT

Create and return a component manager for this device.

Raises
`NotImplementedError` – because it is not implemented.

`init_command_objects()` → None

Set up the command objects.

`activationTime()` → float

Read the time of activation in seconds since Unix epoch.

Returns
Time of activation in seconds since Unix epoch.

`assignedResources()` → list[str]

Read the resources assigned to the device.

The list of resources assigned to the subarray.

Returns
Resources assigned to the device.

`configuredCapabilities()` → list[str]

Read capabilities configured in the Subarray.

A list of capability types with no. of instances in use on this subarray; e.g. Correlators:512, PssBeams:4 PstBeams:4, VlbiBeams:0.

Returns
A list of capability types with no. of instances used in the Subarray

`is_AssignResources_allowed()` → bool

Return whether the *AssignResource* command may be called in the current state.

Raises
`StateModelError` – command not permitted in observation state

Returns
whether the command may be called in the current device state

`is_ReleaseResources_allowed()` → bool

Return whether the *ReleaseResources* command may be called in current state.

Raises
`StateModelError` – command not permitted in observation state

Returns
whether the command may be called in the current device state

`is_ReleaseAllResources_allowed()` → bool

Return whether *ReleaseAllResources* may be called in the current device state.

Raises
`StateModelError` – command not permitted in observation state

Returns
whether the command may be called in the current device state

`is_Configure_allowed()` → bool

Return whether *Configure* may be called in the current device state.

Raises
`StateModelError` – command not permitted in observation state

Returns
whether the command may be called in the current device state

`is_Scan_allowed()` → bool

Return whether the *Scan* command may be called in the current device state.

Raises
`StateModelError` – command not permitted in observation state

Returns
whether the command may be called in the current device state

`is_EndScan_allowed()` → bool

Return whether the *EndScan* command may be called in the current device state.

Raises
`StateModelError` – command not permitted in observation state

Returns
whether the command may be called in the current device state

`is_End_allowed()` → bool

Return whether the *End* command may be called in the current device state.

Raises
`StateModelError` – command not permitted in observation state

Returns
whether the command may be called in the current device state

is_Abort_allowed() → bool

Return whether the *Abort* command may be called in the current device state.

Raises

StateModelError – command not permitted in observation state

Returns

whether the command may be called in the current device state

is_ObsReset_allowed() → bool

Return whether the *ObsReset* command may be called in the current device state.

Raises

StateModelError – command not permitted in observation state

Returns

whether the command may be called in the current device state

is_Restart_allowed() → bool

Return whether the *Restart* command may be called in the current device state.

Raises

StateModelError – command not permitted in observation state

Returns

whether the command may be called in the current device state

main(**args: str*, ***kwargs: str*) → int

Entry point for module.

Parameters

- **args** – positional arguments
- **kwargs** – named arguments

Returns

exit code

2.6 Testing subpackage

This subpackage implements code for testing ska-tango-base devices.

2.6.1 Reference Testing Devices

This subpackage implements reference component managers.

These are example component managers for use in testing, and as explanatory material.

Reference Base Component Manager

This module provided reference implementations of a BaseComponentManager.

It is provided for explanatory purposes, and to support testing of this package.

class FakeBaseComponent(*time_to_return: float = 0.05, time_to_complete: float = 0.4, power: PowerState = PowerState.OFF, fault: Optional[bool] = None, **state_kwargs: Any*)

A fake component for the component manager to work with.

NOTE: There is usually no need to implement a component object. The "component" is an element of the external system under control, such as a piece of hardware or an external service. The component manager object communicates with the component in order to monitor and control it.

This is a very simple fake component with a power state and a fault state. When either of these aspects of state changes, it lets the component manager know by calling its *state_change_callback*.

It can be directly controlled via *off()*, *standby()*, *on()* and *reset()* methods. For testing purposes, it can also be told to simulate a spontaneous state change via simulate_power_state` and *simulate_fault* methods.

When one of these command method is invoked, the component simulates communications latency by sleeping for a short time. It then returns, but simulates any asynchronous work it needs to do by delaying updating task and component state for a short time.

set_state_change_callback(*state_change_callback: Optional[Callable[[...], None]]*) → None

Set a callback to be called when the state of this component changes.

Parameters

state_change_callback – a callback to be call when the state of the component changes

off(*task_callback: Callable, task_abort_event: Event*) → None

Turn the component off.

Parameters

- **task_callback** – a callback to be called whenever the status of this task changes.
- **task_abort_event** – a threading.Event that can be checked for whether this task has been aborted.

standby(*task_callback: Callable, task_abort_event: Event*) → None

Put the component into low-power standby mode.

Parameters

- **task_callback** – a callback to be called whenever the status of this task changes.
- **task_abort_event** – a threading.Event that can be checked for whether this task has been aborted.

on(*task_callback: Callable, task_abort_event: Event*) → None

Turn the component on.

Parameters

- **task_callback** – a callback to be called whenever the status of this task changes.
- **task_abort_event** – a threading.Event that can be checked for whether this task has been aborted.

`simulate_power_state`(*power_state: PowerState*) → None

Simulate a change in component power state.

This could occur as a result of the Off command, or because of some external event/action.

Parameters
`power_state` – the power state

`reset`(*task_callback: Callable*, *task_abort_event: Event*) → None

Reset the component (from fault state).

Parameters

- `task_callback` – a callback to be called whenever the status of this task changes.
- `task_abort_event` – a threading.Event that can be checked for whether this task has been aborted.

`simulate_fault`(*fault_state: bool*) → None

Tell the component to simulate (or stop simulating) a fault.

Parameters
`fault_state` – whether faulty or not.

`property faulty: bool`

Return whether this component is faulty.

Returns
whether this component is faulty.

`property power_state: PowerState`

Return the power state of this component.

Returns
the power state of this component.

`class ReferenceBaseComponentManager`(*logger: Logger*, *communication_state_callback: Callable[[CommunicationStatus], None]*, *component_state_callback: Callable[[], None]*, **args: Any*, *_component: Optional[ComponentT] = None*, ***kwargs: Any*)

A component manager for Tango devices.

It supports:

- Maintaining a connection to its component
- Controlling its component via commands like Off(), Standby(), On(), etc.
- Monitoring its component, e.g. detect that it has been turned off or on

The current implementation is intended to

- illustrate the model
- enable testing of these base classes

It should not generally be used in concrete devices; instead, write a component manager specific to the component managed by the device.

`start_communicating`() → None

Establish communication with the component, then start monitoring.

`stop_communicating`() → None

Break off communication with the component.

`simulate_communication_failure`(*fail_communicate: bool*) → None

Simulate (or stop simulating) a failure to communicate with the component.

Parameters
`fail_communicate` – whether the connection to the component is failing

`property power_state: PowerState`

Power mode of the component.

This is just a bit of syntactic sugar for *self.component_state["power"]*.

Returns
the power mode of the component

`property fault_state: bool`

Whether the component is currently faulting.

Returns
whether the component is faulting

`off`(*task_callback: Optional[Callable[[], None]] = None*) → Tuple[*TaskStatus*, str]

Turn the component off.

Parameters
`task_callback` – a callback to be called whenever the status of this task changes.

Returns
TaskStatus and message

`standby`(*task_callback: Optional[Callable[[], None]] = None*) → Tuple[*TaskStatus*, str]

Put the component into low-power standby mode.

Parameters
`task_callback` – a callback to be called whenever the status of this task changes.

Returns
TaskStatus and message

`on`(*task_callback: Optional[Callable[[], None]] = None*) → Tuple[*TaskStatus*, str]

Turn the component on.

Parameters
`task_callback` – a callback to be called whenever the status of this task changes.

Returns
TaskStatus and message

`reset`(*task_callback: Optional[Callable[[], None]] = None*) → Tuple[*TaskStatus*, str]

Reset the component (from fault state).

Parameters
`task_callback` – a callback to be called whenever the status of this task changes.

Returns
TaskStatus and message

Reference Subarray Component Manager

This module models component management for SKA subarray devices.

`class FakeSubarrayComponent`(*capability_types: list[str]*, *time_to_return: float = 0.05*, *time_to_complete: float = 0.4*, *power: PowerState = PowerState.OFF*, *fault: Optional[bool] = None*, *resourced: bool = False*, *configured: bool = False*, *scanning: bool = False*, *obsfault: bool = False*, ***kwargs: Any*)

A fake component for the component manager to work with.

NOTE: There is usually no need to implement a component object. The "component" is an element of the external system under control, such as a piece of hardware or an external service. In the case of a subarray device, the "component" is likely a collection of Tango devices responsible for monitoring and controlling the various resources assigned to the subarray. The component manager should be written so that it interacts with those Tango devices. But here, we fake up a "component" object to interact with instead.

It supports the *assign*, *release*, *release_all*, *configure*, *scan*, *end_scan*, *end*, *abort*, *obsreset* and *restart* methods. For testing purposes, it can also be told to simulate a spontaneous obs_state change via simulate_power_state` and *simulate_fault* methods.

When one of these command method is invoked, the component simulates communications latency by sleeping for a short time. It then returns, but simulates any asynchronous work it needs to do by delaying updating task and component state for a short time.

`property configured_capabilities: list[str]`

Return the configured capabilities of this component.

Returns

list of strings indicating number of configured instances of each capability type

`assign`(*resources: set[str]*, *task_callback: Callable*, *task_abort_event: threading.Event*) → None

Assign resources.

Parameters

- **resources** – the resources to be assigned.
- **task_callback** – a callback to be called whenever the status of this task changes.
- **task_abort_event** – a threading.Event that can be checked for whether this task has been aborted.

`release`(*resources: set[str]*, *task_callback: Callable*, *task_abort_event: threading.Event*) → None

Release resources.

Parameters

- **resources** – resources to be released
- **task_callback** – a callback to be called whenever the status of this task changes.
- **task_abort_event** – a threading.Event that can be checked for whether this task has been aborted.

`release_all`(*task_callback: Callable*, *task_abort_event: Event*) → None

Release all resources.

Parameters

- **task_callback** – a callback to be called whenever the status of this task changes.
- **task_abort_event** – a threading.Event that can be checked for whether this task has been aborted.

configure(*configuration: dict*, *task_callback: Callable*, *task_abort_event: Event*) → None

Configure the component.

Parameters

- **configuration** – the configuration to be configured
- **task_callback** – a callback to be called whenever the status of this task changes.
- **task_abort_event** – a threading.Event that can be checked for whether this task has been aborted.

deconfigure(*task_callback: Callable*, *task_abort_event: Event*) → None

Deconfigure this component.

Parameters

- **task_callback** – a callback to be called whenever the status of this task changes.
- **task_abort_event** – a threading.Event that can be checked for whether this task has been aborted.

scan(*args: Any*, *task_callback: Callable*, *task_abort_event: Event*) → None

Start scanning.

Parameters

- **args** – positional arguments
- **task_callback** – a callback to be called whenever the status of this task changes.
- **task_abort_event** – a threading.Event that can be checked for whether this task has been aborted.

end_scan(*task_callback: Callable*, *task_abort_event: Event*) → None

End scanning.

Parameters

- **task_callback** – a callback to be called whenever the status of this task changes.
- **task_abort_event** – a threading.Event that can be checked for whether this task has been aborted.

simulate_scan_stopped() → None

Tell the component to simulate spontaneous stopping its scan.

simulate_obsfault(*obsfault: bool*) → None

Tell the component to simulate an obsfault.

Parameters

obsfault – fault indicator

obsreset(*task_callback: Callable*, *task_abort_event: Event*) → None

Reset an observation that has faulted or been aborted.

Parameters

- **task_callback** – a callback to be called whenever the status of this task changes.
- **task_abort_event** – a threading.Event that can be checked for whether this task has been aborted.

`restart`(*task_callback: Callable*, *task_abort_event: Event*) → None

Restart the component after it has faulted or been aborted.

Parameters

- **`task_callback`** – a callback to be called whenever the status of this task changes.
- **`task_abort_event`** – a threading.Event that can be checked for whether this task has been aborted.

`class ReferenceSubarrayComponentManager`(*capability_types: list[str]*, *logger: logging.Logger*, *communication_state_callback: Callable[[CommunicationStatus], None]*, *component_state_callback: Callable[[], None]*, *_component: Optional[FakeSubarrayComponent] = None*)

A component manager for SKA subarray Tango devices.

The current implementation is intended to * illustrate the model * enable testing of the base classes

It should not generally be used in concrete devices; instead, write a subclass specific to the component managed by the device.

`assign`(*resources: set[str]*, *task_callback: Optional[Callable[[], None]] = None*) → tuple[*TaskStatus*, str]

Assign resources to the component.

Parameters

- **`resources`** – resources to be assigned
- **`task_callback`** – a callback to be called whenever the status of this task changes.

Returns

task status and message

`release`(*resources: set[str]*, *task_callback: Optional[Callable[[], None]] = None*) → tuple[*TaskStatus*, str]

Release resources from the component.

Parameters

- **`resources`** – resources to be released
- **`task_callback`** – a callback to be called whenever the status of this task changes.

Returns

task status and message

`release_all`(*task_callback: Optional[Callable[[], None]] = None*) → tuple[*TaskStatus*, str]

Release all resources.

Parameters

`task_callback` – a callback to be called whenever the status of this task changes.

Returns

task status and message

`configure`(*configuration: dict[str, Any]*, *task_callback: Optional[Callable[[], None]] = None*) → tuple[*TaskStatus*, str]

Configure the component.

Parameters

- **`configuration`** – the configuration to be configured
- **`task_callback`** – a callback to be called whenever the status of this task changes.

Returns
task status and message

`deconfigure`(*task_callback: Optional[Callable[[], None]] = None*) → tuple[*TaskStatus*, str]

Deconfigure this component.

Parameters
`task_callback` – a callback to be called whenever the status of this task changes.

Returns
task status and message

`scan`(*args: Any*, *task_callback: Optional[Callable[[], None]] = None*) → tuple[*TaskStatus*, str]

Start scanning.

Parameters

- `args` – additional arguments
- `task_callback` – a callback to be called whenever the status of this task changes.

Returns
task status and message

`end_scan`(*task_callback: Optional[Callable[[], None]] = None*) → tuple[*TaskStatus*, str]

End scanning.

Parameters
`task_callback` – a callback to be called whenever the status of this task changes.

Returns
task status and message

`abort`(*task_callback: Optional[Callable[[], None]] = None*) → tuple[*TaskStatus*, str]

Tell the component to abort the observation.

Parameters
`task_callback` – a callback to be called whenever the status of this task changes.

Returns
task status and message

`obsreset`(*task_callback: Optional[Callable[[], None]] = None*) → tuple[*TaskStatus*, str]

Deconfigure the component but do not release resources.

Parameters
`task_callback` – a callback to be called whenever the status of this task changes.

Returns
task status and message

`restart`(*task_callback: Optional[Callable[[], None]] = None*) → tuple[*TaskStatus*, str]

Tell the component to restart.

It will return to a state in which it is unconfigured and empty of assigned resources.

Parameters
`task_callback` – a callback to be called whenever the status of this task changes.

Returns
task status and message

`property assigned_resources: list[str]`

Return the resources assigned to the component.

Returns
the resources assigned to the component

`property configured_capabilities: list[str]`

Return the configured capabilities of the component.

Returns
list of strings indicating number of configured instances of each capability type

2.7 Alarm Handler Device

This module implements SKAAlarmHandler, a generic base device for Alarms for SKA.

It exposes SKA alarms and SKA alerts as Tango attributes. SKA Alarms and SKA/Element Alerts are rules-based configurable conditions that can be defined over multiple attribute values and quality factors, and are separate from the "built-in" Tango attribute alarms.

`class AlarmHandlerComponentManager`(*logger: Logger*, *communication_state_callback: Optional[Callable[[CommunicationStatus], None]] = None*, *component_state_callback: Optional[Callable[[...], None]] = None*, ***state: Any*)

A stub for an alarm handler component manager.

`class SKAAlarmHandler`(**args: Any*, ***kwargs: Any*)

A generic base device for Alarms for SKA.

`init_command_objects()` → None

Set up the command objects.

`create_component_manager()` → ComponentManagerT

Create and return a component manager for this device.

Raises
`NotImplementedError` – because it is not implemented.

`statsNrAlerts()` → int

Read number of active alerts.

Returns
Number of active alerts

`statsNrAlarms()` → int

Read number of active alarms.

Returns
Number of active alarms

`statsNrNewAlarms()` → int

Read number of new active alarms.

Returns
Number of new active alarms

`statsNrUnackAlarms()` → float

Read number of unacknowledged alarms.

Returns
Number of unacknowledged alarms.

`statsNrRtnAlarms()` → float

Read number of returned alarms.

Returns
Number of returned alarms

`activeAlerts()` → list[str]

Read list of active alerts.

Returns
List of active alerts

`activeAlarms()` → list[str]

Read list of active alarms.

Returns
List of active alarms

`class GetAlarmRuleCommand`(*logger: Optional[Logger] = None*)

A class for the SKAAlarmHandler's GetAlarmRule() command.

`do`(**args: Any*, ***kwargs: Any*) → str

Stateless hook for SKAAlarmHandler GetAlarmRule() command.

Parameters
- **`args`** – positional arguments to the command. This command takes a single positional argument, which is the name of the alarm.
- **`kwargs`** – keyword arguments to the command. This command does not take any, so this should be empty.

Returns
JSON string containing alarm configuration info: rule, actions, etc.

`class GetAlarmDataCommand`(*logger: Optional[Logger] = None*)

A class for the SKAAlarmHandler's GetAlarmData() command.

`do`(**args: Any*, ***kwargs: Any*) → str

Stateless hook for SKAAlarmHandler GetAlarmData() command.

Parameters
- **`args`** – positional arguments to the command. This command takes a single positional argument, which is the name of the alarm.
- **`kwargs`** – keyword arguments to the command. This command does not take any, so this should be empty.

Returns
JSON string specifying alarm data

`class GetAlarmAdditionalInfoCommand`(*logger: Optional[Logger] = None*)

A class for the SKAAlarmHandler's GetAlarmAdditionalInfo() command.

`do`(**args: Any*, ***kwargs: Any*) → str

Stateless hook for SKAAlarmHandler GetAlarmAdditionalInfo() command.

Parameters
- **`args`** – positional arguments to the command. This command takes a single positional argument, which is the name of the alarm.

- **kwargs** – keyword arguments to the command. This command does not take any, so this should be empty.

Returns
JSON string specifying alarm additional info

`class GetAlarmStatsCommand`(*logger: Optional[Logger] = None*)

A class for the SKAAlarmHandler's GetAlarmStats() command.

`do`(**args: Any*, ***kwargs: Any*) → str

Stateless hook for SKAAlarmHandler GetAlarmStats() command.

Parameters

- **args** – positional arguments to the command. This command does not take any, so this should be empty.
- **kwargs** – keyword arguments to the command. This command does not take any, so this should be empty.

Returns
JSON string specifying alarm stats

`class GetAlertStatsCommand`(*logger: Optional[Logger] = None*)

A class for the SKAAlarmHandler's GetAlertStats() command.

`do`(**args: Any*, ***kwargs: Any*) → str

Stateless hook for SKAAlarmHandler GetAlertStats() command.

Parameters

- **args** – positional arguments to the command. This command does not take any, so this should be empty.
- **kwargs** – keyword arguments to the command. This command does not take any, so this should be empty.

Returns
JSON string specifying alert stats

`main`(**args: str*, ***kwargs: str*) → int

Entry point for module.

Parameters

- **args** – positional arguments
- **kwargs** – named arguments

Returns
exit code

2.8 Capability Device

SKACapability.

Capability handling device

`class CapabilityComponentManager`(*logger: Logger*, *communication_state_callback: Optional[Callable[[CommunicationStatus], None]] = None*, *component_state_callback: Optional[Callable[[...], None]] = None*, ***state: Any*)

A stub for an SKA capability component manager.

`class SKACapability`(**args: Any*, ***kwargs: Any*)

A Capability handling device.

It exposes the instances of configured capabilities.

`init_command_objects`() → None

Set up the command objects.

`class InitCommand`(*device: tango.server.Device*, *logger: Optional[Logger] = None*)

A class for the CapabilityDevice's init_device() "command".

`do`(**args: Any*, ***kwargs: Any*) → tuple[*ResultCode*, str]

Stateless hook for device initialisation.

Parameters

- **args** – positional arguments to the command. This command does not take any, so this should be empty.
- **kwargs** – keyword arguments to the command. This command does not take any, so this should be empty.

Returns

A tuple containing a return code and a string message indicating status. The message is for information purpose only.

`create_component_manager`() → ComponentManagerT

Create and return a component manager for this device.

Raises

`NotImplementedError` – because it is not implemented.

`activationTime`() → float

Read time of activation since Unix epoch.

Returns

Activation time in seconds

`configuredInstances`() → int

Read the number of instances of a capability in the subarray.

Returns

The number of configured instances of a capability in a subarray

`usedComponents`() → list[str]

Read the list of components with no.

of instances in use on this Capability

Returns

The number of components currently in use.

`class ConfigureInstancesCommand`(*device: SKACapability*, *logger: Optional[Logger] = None*)

A class for the SKALoggerDevice's SetLoggingLevel() command.

`do`(**args: Any*, ***kwargs: Any*) → tuple[*ResultCode*, str]

Stateless hook for ConfigureInstances()) command functionality.

Parameters

- **args** – positional arguments to the command. This command takes a single integer number of instances.
- **kwargs** – keyword arguments to the command. This command does not take any, so this should be empty.

Returns
A tuple containing a return code and a string message indicating status. The message is for information purpose only.

`main`(**args: str*, ***kwargs: str*) → int

Entry point for module.

Parameters

- **`args`** – positional arguments
- **`kwargs`** – named arguments

Returns
exit code

2.9 Logger Device

This module implements SKALogger device, a generic base device for logging for SKA.

It enables to view on-line logs through the Tango Logging Services and to store logs using Python logging. It configures the log levels of remote logging for selected devices.

`class LoggerComponentManager`(*logger: Logger*, *communication_state_callback: Optional[Callable[[CommunicationStatus], None]] = None*, *component_state_callback: Optional[Callable[[...], None]] = None*, ***state: Any*)

A stub for an logger component manager.

`class SKALogger`(**args: Any*, ***kwargs: Any*)

A generic base device for Logging for SKA.

`init_command_objects`() → None

Set up the command objects.

`create_component_manager`() → ComponentManagerT

Create and return the component manager for this device.

Returns
None, this device doesn't have a component manager

`class SetLoggingLevelCommand`(*logger: Optional[Logger] = None*)

A class for the SKALoggerDevice's SetLoggingLevel() command.

`do`(**args: Any*, ***kwargs: Any*) → Tuple[*ResultCode*, str]

Stateless hook for SetLoggingLevel() command functionality.

Parameters

- **`args`** – positional arguments to the command. This command takes a single positional argument, which is a tuple consisting of list of logging levels and list of tango devices.
- **`kwargs`** – keyword arguments to the command. This command does not take any, so this should be empty.

Returns
A tuple containing a return code and a string message indicating status. The message is for information purpose only.

`main`(**args: str*, ***kwargs: str*) → int

Entry point for module.

Parameters

- **args** – positional arguments
- **kwargs** – named arguments

Returns

exit code

2.10 Controller Device

SKAController.

Controller device

class ControllerComponentManager(*logger: Logger*, *communication_state_callback: Optional[Callable[[CommunicationStatus], None]] = None*, *component_state_callback: Optional[Callable[[...], None]] = None*, ***state: Any*)

A stub for an controller component manager.

class SKAController(**args: Any*, ***kwargs: Any*)

Controller device.

`init_command_objects`() → None

Set up the command objects.

class InitCommand(*device: tango.server.Device*, *logger: Optional[Logger] = None*)

A class for the SKAController's init_device() "command".

`do`(**args: Any*, ***kwargs: Any*) → tuple[ResultCode, str]

Stateless hook for device initialisation.

Parameters

- **args** – positional arguments to the command. This command does not take any, so this should be empty.
- **kwargs** – keyword arguments to the command. This command does not take any, so this should be empty.

Returns

A tuple containing a return code and a string message indicating status. The message is for information purpose only.

`create_component_manager`() → ComponentManagerT

Create and return a component manager for this device.

Raises

`NotImplementedError` – because it is not implemented.

`elementLoggerAddress`() → str

Read FQDN of Element Logger device.

Returns

FQDN of Element Logger device

elementAlarmAddress() → str

Read FQDN of Element Alarm device.

Returns
FQDN of Element Alarm device

elementTelStateAddress() → str

Read FQDN of Element TelState device.

Returns
FQDN of Element TelState device

elementDatabaseAddress() → str

Read FQDN of Element Database device.

Returns
FQDN of Element Database device

maxCapabilities() → list[str]

Read maximum number of instances of each capability type.

Returns
list of maximum number of instances of each capability type

availableCapabilities() → list[str]

Read list of available number of instances of each capability type.

Returns
list of available number of instances of each capability type

class IsCapabilityAchievableCommand(*device:* SKAController, *logger: Optional[Logger] = None*)

A class for the SKAController's IsCapabilityAchievable() command.

do(**args: Any*, ***kwargs: Any*) → bool

Stateless hook for device IsCapabilityAchievable() command.

Parameters

- **args** – positional arguments to the command. There is a single positional argument: an array consisting of pairs of * [nrInstances]: DevLong. Number of instances of the capability. * [Capability types]: DevString. Type of capability.
- **kwargs** – keyword arguments to the command. This command does not take any, so this should be empty.

Returns
Whether the capability is achievable

main(**args: str*, ***kwargs: str*) → int

Entry point for module.

Parameters

- **args** – positional arguments
- **kwargs** – named arguments

Returns
exit code

2.11 Tel State Device

SKATelState.

A generic base device for Telescope State for SKA.

class TelStateComponentManager(*logger: Logger*, *communication_state_callback: Optional[Callable[[CommunicationStatus], None]] = None*, *component_state_callback: Optional[Callable[[...], None]] = None*, ***state: Any*)

A stub for a telescope state component manager.

class SKATelState(**args: Any*, ***kwargs: Any*)

A generic base device for Telescope State for SKA.

create_component_manager() → ComponentManagerT

Create and return a component manager for this device.

Raises

`NotImplementedError` – because it is not implemented.

main(**args: str*, ***kwargs: str*) → int

Entry point for module.

Parameters

- **args** – positional arguments
- **kwargs** – named arguments

Returns

exit code

2.12 Commands

This module provides abstract base classes for device commands, and a ResultCode enum.

The following command classes are provided:

- **FastCommand**: implements the common pattern for fast commands; that is, commands that do not perform any blocking action. These commands call their callback to indicate that they have started, then execute their do hook, and then immediately call their callback to indicate that they have completed.
- **DeviceInitCommand: Implements the common pattern for device Init**
  commands. This is just a FastCommands, with a fixed signature for the `__init__` method.
- **SlowCommand**: implements the common pattern for slow commands; that is, commands that need to perform a blocking action, such as file I/O, network I/O, waiting for a shared resource, etc. These commands call their callback to indicate that they have started, then execute their do hook. However they do not immediately call their callback to indicate completion. They assume that the do hook will launch work in an asynchronous context (such as a thread), and make it the responsibility of that asynchronous context to call the command's `completed` method when it finishes.
- **SubmittedSlowCommand: `whereas SlowCommand` makes no assumptions**
  about how the command will be implemented, `SubmittedSlowCommand` assumes the current device structure: i.e. a command tracker, and a component manager with support for submitting tasks.

`class ResultCode`(*value*)

Python enumerated type for command result codes.

`OK = 0`

The command was executed successfully.

`STARTED = 1`

The command has been accepted and will start immediately.

`QUEUED = 2`

The command has been accepted and will be executed at a future time.

`FAILED = 3`

The command could not be executed.

`UNKNOWN = 4`

The status of the command is not known.

`REJECTED = 5`

The command execution has been rejected.

`NOT_ALLOWED = 6`

The command is not allowed to be executed.

`ABORTED = 7`

The command in progress has been aborted.

`class FastCommand`(*logger: Optional[Logger] = None*)

An abstract class for Tango device server commands that execute quickly.

That is, they do not perform any blocking operation, so can be safely run synchronously.

`class SlowCommand`(*callback: Optional[Callable], logger: Optional[Logger] = None*)

An abstract class for Tango device server commands that execute slowly.

That is, they perform at least one blocking operation, such as file I/O, network I/O, waiting for a shared resources, etc. They therefore need to be run asynchronously in order to preserve throughput.

`class DeviceInitCommand`(*device: tango.server.Device, logger: Optional[Logger] = None*)

A `SlowCommand` with a fixed initialisation interface.

Although most commands have lots of flexibility in how they are initialised, device `InitCommand` instances are always called in the same way. This class fixes that interface. `InitCommand` instances should inherit from this command, rather than directly from `SlowCommand`, to ensure that their initialisation signature is correct.

`class SubmittedSlowCommand`(*command_name: str, command_tracker: CommandTrackerProtocol, component_manager: BaseComponentManager, method_name: str, callback: Optional[Callable] = None, logger: Optional[Logger] = None*)

A SlowCommand with lots of implementation-dependent boilerplate in it.

Whereas the SlowCommand is generic, and makes no assumptions about how the slow command will be executed, this SubmittedSlowCommand contains implementation-dependent information about the SKABaseDevice model, such as knowledge of the command tracker and component manager. It thus implements a lot of boilerplate code, and allows us to avoid implementing many identical commands.

Parameters

- **`command_name`** – name of the command e.g. "Scan". This is only used to ensure that the generated command id contains it.

- **command_tracker** – the device's command tracker
- **component_manager** – the device's component manager
- **method_name** – name of the component manager method to be invoked by the do hook
- **callback** – an optional callback to be called when this command starts and finishes.
- **logger** – a logger for this command to log with.

`do`(**args: Any*, ***kwargs: Any*) → tuple[*ResultCode*, str]

Stateless hook for command functionality.

Parameters

- **args** – positional args to the component manager method
- **kwargs** – keyword args to the component manager method

Returns

A tuple containing the task status (e.g. QUEUED or REJECTED), and a string message containing a command_id (if the command has been accepted) or an informational message (if the command was rejected)

`class CommandTrackerProtocol`(**args*, ***kwds*)

All things to do with commands.

`new_command`(*command_name: str*, *completed_callback: Optional[Callable[[], None]] = None*) → str

Create a new command.

Parameters

- **command_name** – the command name
- **completed_callback** – an optional callback for command completion

`update_command_info`(*command_id: str*, *status: Optional[*TaskStatus*] = None*, *progress: Optional[int] = None*, *result: Optional[tuple[*ResultCode*, str]] = None*, *exception: Optional[Exception] = None*) → None

Update status information on the command.

Parameters

- **command_id** – the unique command id
- **status** – the status of the asynchronous task
- **progress** – the progress of the asynchronous task
- **result** – the result of the completed asynchronous task
- **exception** – any exception caught in the running task

2.13 Control Model

Module for SKA Control Model (SCM) related code.

For further details see the SKA1 CONTROL SYSTEM GUIDELINES (CS_GUIDELINES MAIN VOLUME) Document number: 000-000000-010 GDL And architectural updates: https://jira.skatelescope.org/browse/ADR-8 https://confluence.skatelescope.org/pages/viewpage.action?pageId=105416556

The enumerated types mapping to the states and modes are included here, as well as other useful enumerations.

2.14 Faults

General SKA Tango Device Exceptions.

`exception SKABaseError`

Base class for all SKA Tango Device exceptions.

`exception GroupDefinitionsError`

Error parsing or creating groups from GroupDefinitions.

`exception LoggingLevelError`

Error evaluating logging level.

`exception LoggingTargetError`

Error parsing logging target string.

`exception ResultCodeError`

A method has returned an invalid return code.

`exception StateModelError`

Error in state machine model related to transitions or state.

`exception CommandError`

Error executing a BaseCommand or similar.

`exception CapabilityValidationError`

Error in validating capability input against capability types.

`exception ComponentError`

Component cannot perform as requested.

`exception ComponentFault`

Component is in FAULT state and cannot perform as requested.

2.15 Release

Release information for ska_tango_base Python Package.

2.16 Utils

General utilities that may be useful to SKA devices and clients.

`exception_manager`(*cls: type[Exception], callback: Optional[Callable] = None*) → Generator

Return a context manager that manages exceptions.

Parameters

- `cls` – class type
- `callback` – a callback

Yields

return a context manager

`get_dev_info`(*domain_name: str, device_server_name: str, device_ref: str*) → tango.DbDevInfo

Get device info.

Parameters

- `domain_name` – tango domain name
- `device_server_name` – tango device server name
- `device_ref` – tango device reference

Returns

database device info instance

`dp_set_property`(*device_name: str, property_name: str, property_value: Any*) → None

Use a DeviceProxy to set a device property.

Parameters

- `device_name` – tango device name
- `property_name` – tango property name
- `property_value` – tango property value

`get_device_group_and_id`(*device_name: str*) → list[str]

Return the group and id part of a device name.

Parameters

`device_name` – tango device name

Returns

group & id part of tango device name

`convert_api_value`(*param_dict: dict[str, str]*) → tuple[str, Any]

Validate tango command parameters which are passed via json.

Parameters

`param_dict` – parameters

Raises

`Exception` – invalid type

Returns

tuple(name, value)

`coerce_value`(*value: tango.DevState*) → str

Coerce tango.DevState values to string, leaving other values alone.

Parameters
`value` – a tango DevState

Returns
DevState as a string

`get_dp_attribute`(*device_proxy: tango.DeviceProxy*, *dp_attribute: tango.server.attribute*, *with_value: bool = False*, *with_context: bool = False*) → dict

Get an attribute from a DeviceProxy.

:param device_proxy:a tango device proxy :param dp_attribute: Attribute :param with_value: default False :param with_context: default False

Returns
dictionary of attribute info

`get_dp_command`(*device_name: str*, *dp_command: tango.server.command*, *with_context: bool = False*) → dict

Get a command from a DeviceProxy.

Parameters

- `device_name` – tango device name
- `dp_command` – tango command
- `with_context` – default False

Returns
dictionary of command info

`get_tango_device_type_id`(*tango_address: str*) → list[str]

Return the type id of a TANGO device.

Parameters
`tango_address` – tango device address

Returns
the type id of the device

`get_groups_from_json`(*json_definitions: List[str]*) → dict

Return a dict of tango.Group objects matching the JSON definitions.

Extracts the definitions of groups of devices and builds up matching tango.Group objects. Some minimal validation is done - if the definition contains nothing then None is returned, otherwise an exception will be raised on error.

This function will *NOT* attempt to verify that the devices exist in the Tango database, nor that they are running.

The definitions would typically be provided by the Tango device property "GroupDefinitions", available in the SKABaseDevice. The property is an array of strings. Thus a sequence is expected for this function.

Each string in the list is a JSON serialised dict defining the "group_name", "devices" and "subgroups" in the group. The tango.Group() created enables easy access to the managed devices in bulk, or individually. Empty and whitespace-only strings will be ignored.

The general format of the list is as follows, with optional "devices" and "subgroups" keys:

```
[
    {"group_name": "<name>", "devices": ["<dev name>", ...]},
    {
        "group_name": "<name>",
        "devices": ["<dev name>", "<dev name>", ...],
        "subgroups" : [{<nested group>}, {<nested group>}, ...]
    },
    ...
]
```

For example, a hierarchy of racks, servers and switches:

```
[
    {
        "group_name": "servers",
        "devices": [
            "elt/server/1", "elt/server/2", "elt/server/3", "elt/server/4"
        ]
    },
    {
        "group_name": "switches",
        "devices": ["elt/switch/A", "elt/switch/B"]
    },
    {
        "group_name": "pdus",
        "devices": ["elt/pdu/rackA", "elt/pdu/rackB"]
    },
    {
        "group_name": "racks",
        "subgroups": [
            {
                "group_name": "rackA",
                "devices": [
                    "elt/server/1", "elt/server/2", "elt/switch/A", "elt/pdu/rackA"
                ]
            },
            {
                "group_name": "rackB",
                "devices": [
                    "elt/server/3",
                    "elt/server/4",
                    "elt/switch/B",
                    "elt/pdu/rackB"
                ],
                "subgroups": []
            }
        ]
    }
]
```

Parameters

json_definitions – Sequence of strings, each one a JSON dict with keys "group_name", and one or both of: "devices" and "subgroups", recursively defining the hierarchy.

Returns
A dictionary, the keys of which are the names of the groups, in the following form: {"<group name 1>": <tango.Group>, "<group name 2>": <tango.Group>, . . . }. Will be an empty dict if no groups were specified.

Raises
GroupDefinitionsError – # noqa DAR401,DAR402 arising from GroupDefinitionsError - If error parsing JSON string. - If missing keys in the JSON definition. - If invalid device name. - If invalid groups included. - If a group has multiple parent groups. - If a device is included multiple time in a hierarchy. E.g. g1:[a,b] g2:[a,c] g3:[g1,g2]

validate_capability_types(*command_name: str, requested_capabilities: list[str], valid_capabilities: list[str]*) → None

Check the validity of the capability types passed to the specified command.

Parameters

- **command_name** – The name of the command to be executed.
- **requested_capabilities** – A list of strings representing capability types.
- **valid_capabilities** – A list of strings representing capability types.

validate_input_sizes(*command_name: str, argin: Tuple[List[int], List[str]]*) → None

Check the validity of the input parameters passed to the specified command.

Parameters

- **command_name** – The name of the command which is to be executed.
- **argin** – A tuple of two lists

convert_dict_to_list(*dictionary: dict[Any, Any]*) → list[str]

Convert a dictionary to a list of "key:value" strings.

Parameters
dictionary – a dictionary to be converted

Returns
a list of key/value strings

for_testing_only(*func: ~typing.Callable, _testing_check: ~typing.Callable[[], bool] = <function <lambda>>*) → Callable

Return a function that warns if called outside of testing, then calls a function.

This is intended to be used as a decorator that marks a function as available for testing purposes only. If the decorated function is called outside of testing, a warning is raised.

```
@for_testing_only
def _straight_to_state(self, state):
    ...
```

Parameters

- **func** – function to be wrapped
- **_testing_check** – True if testing

Returns
the wrapper function

`generate_command_id`(*command_name: str*) → str

Generate a unique command ID for a given command name.

Parameters
`command_name` – name of the command for which an ID is to be generated.

Returns
a unique command ID string

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX