SKA Tango Base Documentation

Release 0.11.4

NCRA India

Sep 06, 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 EngageSKA Nexus 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.

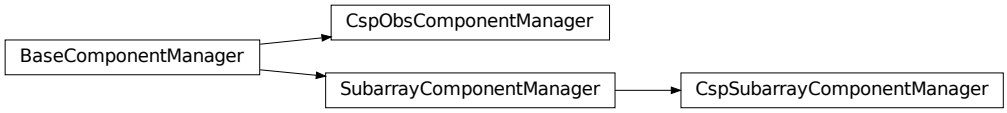
A *component manager* provides for monitoring and control of a component. It is *highly recommended* to implement and thoroughly test your component manager *before* embedding it in a Tango device.

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.

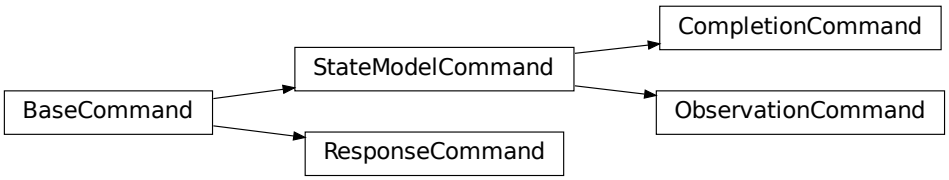
1. **Do you really need to implement the command?** If the command to be implemented is part of the Tango device you will inherit from, perhaps the current implementation is exactly what you need.

   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.

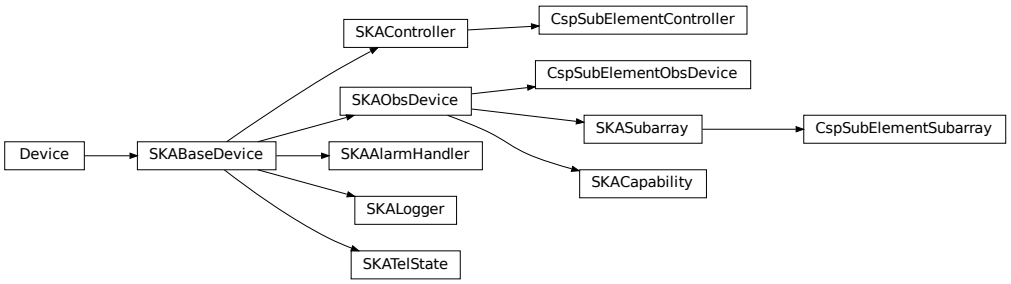


3. **Implement class methods.**
   - In many cases, you only need to implement the `do()` method.
   - To constrain when the command is allowed to be invoked, override the `is_allowed()` method.

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.

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 specifies the admin mode model for SKA LMC Tango devices.

It consists of a single public class: *AdminModeModel*. This uses a state machine to device device adminMode, represented as a *ska_tango_base.control_model.AdminMode* enum value, and reported by Tango devices through the `AdminMode` attribute.

class ska_tango_base.base.admin_mode_model.**AdminModeModel**(*logger*, *callback=None*)

This class implements the state model for device adminMode.

The model supports the five admin modes defined by the values of the *ska_tango_base.control_model.AdminMode* enum.

- **NOT_FITTED**: the component that the device is supposed to monitor is not fitted.
- **RESERVED**: the component that the device is monitoring is not in use because it is redundant to other devices. It is ready to take over should other devices fail.
- **OFFLINE**: the component that the device is monitoring device has been declared by SKA operations not to be used
- **MAINTENANCE**: the component that the device is monitoring cannot be used for science purposes but can be for engineering / maintenance purposes, such as testing, debugging, etc.
- **ONLINE**: the component that the device is monitoring can be used for science purposes.

The admin mode state machine allows for:

- any transition between the modes NOT_FITTED, RESERVED and OFFLINE (e.g. an unfitted device being fitted as a redundant or non-redundant device, a redundant device taking over when another device fails, etc.)
- any transition between the modes OFFLINE, MAINTENANCE and ONLINE (e.g. an online device being taken offline or put into maintenance mode to diagnose a fault, a faulty device moving between maintenance and offline mode as it undergoes sporadic periods of diagnosis.)

The actions supported are:

- **to_not_fitted**
- **to_reserved**

- **to_offline**
- **to_maintenance**
- **to_online**

A diagram of the admin mode model, as designed, is shown below

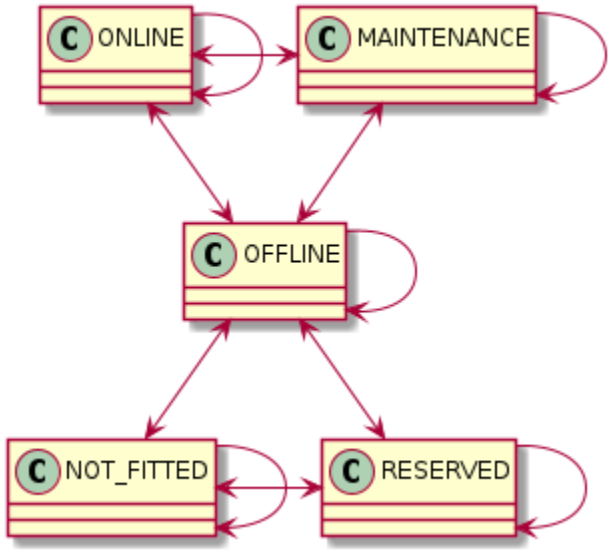
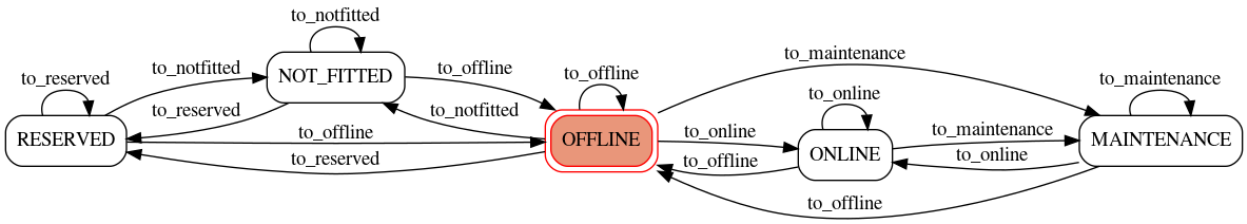


Fig. 1: Diagram of the admin mode model

The following is an diagram of the underlying state machine, automatically generated from the code. Its equivalence to the diagram above demonstrates that the implementation is faithful to the design.



property admin_mode

Return the admin_mode.

Returns

admin_mode of this state model

Return type

AdminMode

is_action_allowed(*action*, *raise_if_disallowed=False*)

Return whether a given action is allowed in the current state.

Parameters

- **action** (*str*) – an action, as given in the transitions table

- **raise_if_disallowed** (*bool*) – whether to raise an exception if the action is disallowed, or merely return False (optional, defaults to False)

Raises
StateModelError – if the action is unknown to the state machine

Returns
whether the action is allowed in the current state

Return type
bool

perform_action(*action*)

Perform an action on the state model.

Parameters
action (*str*) – an action, as given in the transitions table

2.1.2 Op State Model

This module specifies the operational state ("opState") model for SKA LMC Tango devices.

It consists of:

- an underlying state machine: `_OpStateMachine`
- an *OpStateModel* that maps state machine state to device "op state". This "op state" is currently represented as a `tango.DevState` enum value, and reported using the tango device's special `state()` method.

class ska_tango_base.base.op_state_model.**OpStateModel**(*logger*, *callback=None*)

This class implements the state model for device operational state ("opState").

The model supports the following states, represented as values of the `tango.DevState` enum.

- **INIT**: the device is initialising.
- **DISABLE**: the device has been told not to monitor its telescope component
- **UNKNOWN**: the device is monitoring (or at least trying to monitor) its telescope component but is unable to determine the component's state
- **OFF**: the device is monitoring its telescope component and the component is powered off
- **STANDBY**: the device is monitoring its telescope component and the component is in low-power standby mode
- **ON**: the device is monitoring its telescope component and the component is turned on
- **FAULT**: the device is monitoring its telescope component and the component has failed or is in an inconsistent state.

These are essentially the same states as the underlying `_OpStateMachine`, except that all initialisation states are mapped to the INIT DevState.

The actions supported are:

- **init_invoked**: the device has started initialising
- **init_completed**: the device has finished initialising
- **component_disconnected**: the device his disconnected from the telescope component that it is supposed to manage (for example because its admin mode was set to OFFLINE). Note, this action indicates a device-initiated, deliberate disconnect; a lost connection would be indicated by a "component_fault" or "component_unknown" action, depending on circumstances.

- **component_unknown**: the device is unable to determine the state of its component.
- **component_off**: the component has been switched off
- **component_standby**: the component has switched to low-power standby mode
- **component_on**: the component has been switched on

A diagram of the operational state model, as implemented, is shown below.

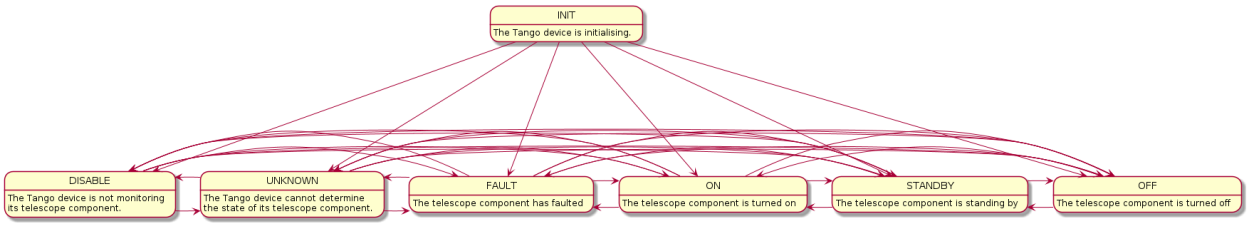


Fig. 2: Diagram of the operational state model

The following hierarchical diagram is more explanatory; however note that the implementation does *not* use a hierarchical state machine.

property op_state

Return the op state.

Returns
the op state of this state model

Return type
tango.DevState

is_action_allowed(*action*, *raise_if_disallowed=False*)

Return whether a given action is allowed in the current state.

Parameters

- **action** (*str*) – an action, as given in the transitions table
- **raise_if_disallowed** (*bool*) – whether to raise an exception if the action is disallowed, or merely return False (optional, defaults to False)

Raises
StateModelError – if the action is unknown to the state machine

Returns
whether the action is allowed in the current state

Return type
bool

perform_action(*action*)

Perform an action on the state model.

Parameters
action (*str*) – an action, as given in the transitions table

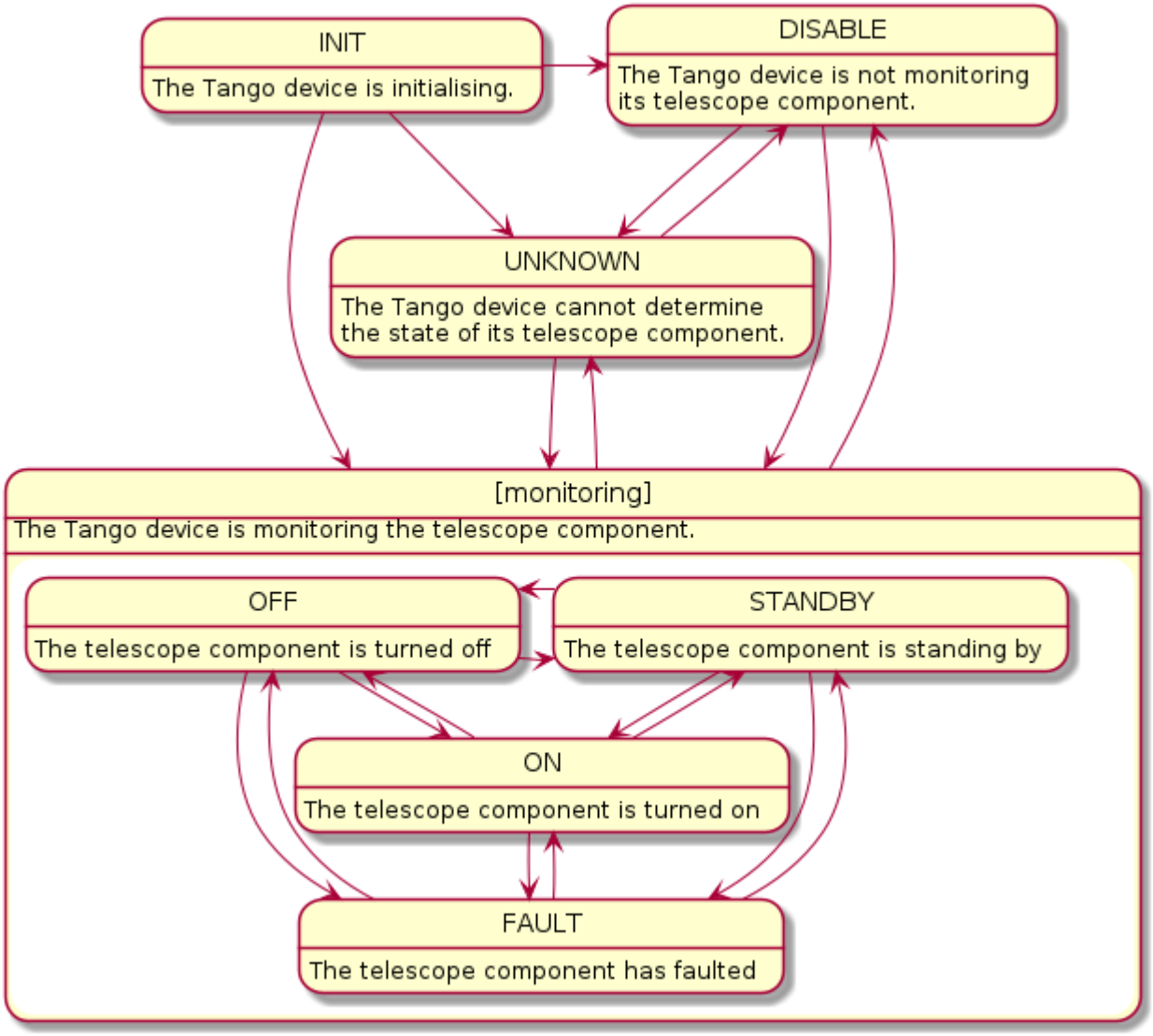


Fig. 3: Diagram of the operational state model

2.1.3 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 a *connection* to 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.

class ska_tango_base.base.component_manager.**BaseComponentManager**(*op_state_model*, **args*, ***kwargs*)

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()

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)

stop_communicating()

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.

property is_communicating

Return whether communication with the component is established.

For example:

- If communication is over a connection, are you connected?
- If communication is via event subscription, are you subscribed, and is the event subsystem healthy?
- If you are polling the component, is the polling loop running, and is the component responsive?

Returns
whether there is currently a connection to the component

Return type
bool

property power_mode

Power mode of the component.

Returns
the power mode of the component

property faulty

Whether the component is currently faulting.

Returns
whether the component is faulting

off()

Turn the component off.

standby()

Put the component into low-power standby mode.

on()

Turn the component on.

reset()

Reset the component (from fault state).

component_power_mode_changed(*power_mode*)

Handle notification that the component's power mode has changed.

This is a callback hook.

Parameters
power_mode (*ska_tango_base.control_model.PowerMode*) – the new power mode of the component

component_fault()

Handle notification that the component has faulted.

This is a callback hook.

2.1.4 Reference Base Component Manager

This module provided a reference implementation of a BaseComponentManager.

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

ska_tango_base.base.reference_component_manager.**check_communicating**(*func*)

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

class ska_tango_base.base.reference_component_manager.**ReferenceBaseComponentManager**(*op_state_model*, **args*, *logger=None*, *_component=None*, ***kwargs*)

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()

Establish communication with the component, then start monitoring.

stop_communicating()

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

property is_communicating

Whether there is currently a connection to the component.

Returns
whether there is currently a connection to the component

Return type
bool

simulate_communication_failure(*fail_communicate*)

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

Parameters
fail_communicate – whether the connection to the component is failing

property power_mode

Power mode of the component.

Returns
the power mode of the component

property faulty

Whether the component is currently faulting.

Returns
whether the component is faulting

off()

Turn the component off.

standby()

Put the component into low-power standby mode.

on()

Turn the component on.

reset()

Reset the component (from fault state).

component_power_mode_changed(*power_mode*)

Handle notification that the component's power mode has changed.

This is a callback hook.

Parameters
power_mode (*ska_tango_base.control_model.PowerMode*) – the new power mode of the component

component_fault()

Handle notification that the component has faulted.

This is a callback hook.

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 ska_tango_base.base.base_device.**SKABaseDevice**(**args: Any*, ***kwargs: Any*)

A generic base device for SKA.

class InitCommand(*target*, *op_state_model*, *logger=None*)

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

do()

Stateless hook for device initialisation.

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

Return type
(*ResultCode*, str)

SkaLevel = tango.server.device_property(dtype=int16, default_value=4)

Device property.

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

GroupDefinitions = tango.server.device_property(dtype=('str',))

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 = tango.server.device_property(dtype=uint16, default_value=4)

Device property.

Default logging level at device startup. See *LoggingLevel*

LoggingTargetsDefault = tango.server.device_property(dtype=DevVarStringArray, default_value=['tango::logger'])

Device property.

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

buildState = tango.server.attribute(dtype=str, doc=Build state of this device)

Device attribute.

versionId = tango.server.attribute(dtype=str, doc=Version Id of this device)

Device attribute.

loggingLevel = tango.server.attribute(dtype=<enum 'LoggingLevel'>, access=tango.AttrWriteType.READ_WRITE, doc=Current logging level for this device - initialises to LoggingLevelDefault on startup)

Device attribute.

See *LoggingLevel*

loggingTargets = tango.server.attribute(dtype=('str',), access=tango.AttrWriteType.READ_WRITE, max_dim_x=4, doc=Logging targets for this device, excluding ska_ser_logging defaults - initialises to LoggingTargetsDefault on startup)

Device attribute.

healthState = tango.server.attribute(dtype=<enum 'HealthState'>, doc=The health state reported for this device. It interprets the current device condition and condition of all managed devices to set this. Most possibly an aggregate attribute.)

Device attribute.

adminMode = tango.server.attribute(dtype=<enum 'AdminMode'>, access=tango.AttrWriteType.READ_WRITE, memorized=True, hw_memorized=True, doc=The admin mode reported for this device. It may interpret the current device condition and condition of all managed devices to set this. Most possibly an aggregate attribute.)

Device attribute.

controlMode = tango.server.attribute(dtype=<enum 'ControlMode'>, access=tango.AttrWriteType.READ_WRITE, memorized=True, hw_memorized=True, doc=The control mode of the device. REMOTE, LOCAL Tango Device accepts only from a 'local' client and ignores commands and queries received from TM or any other 'remote' clients. The Local clients has to release LOCAL control before REMOTE clients can take control again.)

Device attribute.

simulationMode = tango.server.attribute(dtype=<enum 'SimulationMode'>, access=tango.AttrWriteType.READ_WRITE, memorized=True, hw_memorized=True, doc=Reports the simulation mode of the device. Some devices may implement both modes, while others will have simulators that set simulationMode to True while the real devices always set simulationMode to False.)

Device attribute.

testMode = tango.server.attribute(dtype=<enum 'TestMode'>, access=tango.AttrWriteType.READ_WRITE, memorized=True, hw_memorized=True, doc=The test mode of the device. Either no test mode or an indication of the test mode.)

Device attribute.

set_state(*state*)

Set the device state, ensuring that change events are pushed.

Parameters

state (tango.DevState) – the new state

set_status(*status*)

Set the device status, ensuring that change events are pushed.

Parameters
status (*str*) – the new status

init_device()

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.

create_component_manager()

Create and return a component manager for this device.

register_command_object(*command_name*, *command_object*)

Register an object as a handler for a command.

Parameters

- **command_name** (*str*) – name of the command for which the object is being registered
- **command_object** (*Command instance*) – the object that will handle invocations of the given command

get_command_object(*command_name*)

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

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

Returns
the registered command object (handler) for the command

Return type
Command instance

init_command_objects()

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

always_executed_hook()

Perform actions that are executed before every device command.

This is a Tango hook.

delete_device()

Clean up any resources prior to device deletion.

This method is a Tango hook that is called by the device destructor and by the device Init command. It allows for any memory or other resources allocated in the init_device method to be released prior to device deletion.

read_buildState()

Read the Build State of the device.

Returns
the build state of the device

read_versionId()

Read the Version Id of the device.

Returns
the version id of the device

read_loggingLevel()

Read the logging level of the device.

Returns
Logging level of the device.

write_loggingLevel(*value*)

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

read_loggingTargets()

Read the additional logging targets of the device.

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

Returns
Logging level of the device.

write_loggingTargets(*value*)

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

read_healthState()

Read the Health State of the device.

Returns
Health State of the device

read_adminMode()

Read the Admin Mode of the device.

Returns
Admin Mode of the device

Return type
AdminMode

write_adminMode(*value*)

Set the Admin Mode of the device.

Parameters
value (*AdminMode*) – Admin Mode of the device.

Raises
ValueError – for unknown adminMode

read_controlMode()

Read the Control Mode of the device.

Returns
Control Mode of the device

write_controlMode(*value*)

Set the Control Mode of the device.

Parameters
value – Control mode value

read_simulationMode()

Read the Simulation Mode of the device.

Returns
Simulation Mode of the device.

write_simulationMode(*value*)

Set the Simulation Mode of the device.

Parameters
value – SimulationMode

read_testMode()

Read the Test Mode of the device.

Returns
Test Mode of the device

write_testMode(*value*)

Set the Test Mode of the device.

Parameters
value – Test Mode

class GetVersionInfoCommand(*target*, **args*, *logger=None*, ***kwargs*)

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

do()

Stateless hook for device GetVersionInfo() command.

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

Return type
(*ResultCode*, str)

GetVersionInfo()

Return the version information of the device.

To modify behaviour for this command, modify the do() method of the command class.

Returns
Version details of the device.

class ResetCommand(*target*, *op_state_model*, *logger=None*)

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

do()

Stateless hook for device reset.

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

Return type
(*ResultCode*, str)

`is_Reset_allowed()`

Whether the `Reset()` command is allowed to be run in the current state.

Returns
whether the `Reset()` command is allowed to be run in the current state

Return type
boolean

`Reset()`

Reset the device from the FAULT state.

To modify behaviour for this command, modify the do() method of the command class.

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

Return type
(*ResultCode*, str)

`class StandbyCommand(target, op_state_model, logger=None)`

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

`do()`

Stateless hook for Standby() command functionality.

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

Return type
(*ResultCode*, str)

`is_Standby_allowed()`

Check if command Standby is allowed in the current device state.

:raises `CommandError`: if the command is not allowed

Returns
`True` if the command is allowed

Return type
boolean

`Standby()`

Put the device into standby mode.

To modify behaviour for this command, modify the do() method of the command class.

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

Return type
(*ResultCode*, str)

`class OffCommand(target, op_state_model, logger=None)`

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

`do()`

Stateless hook for Off() command functionality.

Returns
A tuple containing a return code and a string message indicating status. The message is for information purpose only.
Return type
(*ResultCode*, str)

`is_Off_allowed()`

Check if command *Off* is allowed in the current device state.

:raises `CommandError`: if the command is not allowed

Returns
`True` if the command is allowed

Return type
boolean

`Off()`

Turn the device off.

To modify behaviour for this command, modify the do() method of the command class.

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

Return type
(*ResultCode*, str)

`class OnCommand(target, op_state_model, logger=None)`

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

`do()`

Stateless hook for On() command functionality.
Returns
A tuple containing a return code and a string message indicating status. The message is for information purpose only.
Return type
(*ResultCode*, str)

`is_On_allowed()`

Check if command *On* is allowed in the current device state.

:raises `CommandError`: if the command is not
allowed

Returns
`True` if the command is allowed

Return type
boolean

`On()`

Turn device on.

To modify behaviour for this command, modify the do() method of the command class.

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

Return type
(*ResultCode*, str)

class DebugDeviceCommand(*target*, **args*, *logger=None*, ***kwargs*)

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

do()

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.

Returns
The TCP port the debugger is listening on.

Return type
DevUShort

start_debugger_and_get_port(*port*)

Start the debugger and return the allocated port.

monkey_patch_all_methods_for_debugger()

Monkeypatch methods that need to be patched for the debugger.

get_all_methods()

Return a list of the device's methods.

static method_must_be_patched_for_debugger(*owner*, *method*)

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 *typing.types.FunctionType* check excludes the Boost methods.

patch_method_for_debugger(*owner*, *name*, *method*)

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.

DebugDevice()

Enable remote debugging of this device.

To modify behaviour for this command, modify the do() method of the command class: *DebugDeviceCommand*.

ska_tango_base.base.base_device.**main**(*args=None*, ***kwargs*)

Launch an SKABaseDevice device.

Parameters

- **args** – positional args to tango.server.run
- **kwargs** – named args to tango.server.run

2.2 Obs subpackage

This subpackage models a SKA Tango observing device.

2.2.1 Obs State Model

This module defines a basic state model for SKA LMC devices that manage observations.

It consists of a single *ObsStateModel* class, which drives a state machine to manage device "obs state", represented by the *ska_tango_base.control_model.ObsState* enum, and published by Tango devices via the `obsState` attribute.

class ska_tango_base.obs.obs_state_model.**ObsStateModel**(*state_machine_factory*, *logger*, *callback=None*)

This class implements the state model for observation state ("obsState").

The model supports states that are values of the *ska_tango_base.control_model.ObsState* enum. Rather than specifying a state machine, it allows a state machine to be provided. Thus, the precise states supported, and the transitions, are not determined in advance.

property obs_state

Return the obs_state.

Returns
obs_state of this state model

Return type
ObsState

is_action_allowed(*action*, *raise_if_disallowed=False*)

Return whether a given action is allowed in the current state.

Parameters

- **action** (*str*) – an action, as given in the transitions table
- **raise_if_disallowed** (*bool*) – whether to raise an exception if the action is disallowed, or merely return False (optional, defaults to False)

Raises
StateModelError – if the action is unknown to the state machine

Returns
whether the action is allowed in the current state

Return type
bool

perform_action(*action*)

Perform an action on the state model.

Parameters
action (*ANY*) – an action, as given in the transitions table

2.2.2 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 ska_tango_base.obs.obs_device.**SKAObsDevice**(**args: Any*, ***kwargs: Any*)

A generic base device for Observations for SKA.

class InitCommand(*target*, *op_state_model*, *logger=None*)

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

do()

Stateless hook for device initialisation.

Returns

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

Return type

(*ResultCode*, str)

obsState = tango.server.attribute(dtype=<enum 'ObsState'>, doc=Observing State)

Device attribute.

obsMode = tango.server.attribute(dtype=<enum 'ObsMode'>, doc=Observing Mode)

Device attribute.

configurationProgress = tango.server.attribute(dtype=uint16, unit=%, max_value=100, min_value=0, doc=Percentage configuration progress)

Device attribute.

configurationDelayExpected = tango.server.attribute(dtype=uint16, unit=seconds, doc=Configuration delay expected in seconds)

Device attribute.

always_executed_hook()

Perform actions that are executed before every device command.

This is a Tango hook.

delete_device()

Clean up any resources prior to device deletion.

This method is a Tango hook that is called by the device destructor and by the device Init command. It allows for any memory or other resources allocated in the init_device method to be released prior to device deletion.

read_obsState()

Read the Observation State of the device.

read_obsMode()

Read the Observation Mode of the device.

read_configurationProgress()

Read the percentage configuration progress of the device.

read_configurationDelayExpected()

Read the expected Configuration Delay in seconds.

ska_tango_base.obs.obs_device.**main**(*args=None*, ***kwargs*)

Launch an SKAObsDevice.

Parameters

- **args** – positional arguments
- **kwargs** – keyword arguments

2.3 CSP subpackage

This subpackage contains base devices specific to CSP.

2.3.1 CSP obs subpackage

This subpackage contains obs device functionality specific to CSP.

CSP Subelement Obs State Model

This module specifies CSP SubElement Observing state machine.

It comprises:

- an underlying state machine: `_CspSubElementObsStateMachine`
- a *CspSubElementObsStateModel* that maps the underlying state machine state to a value of the *ska_tango_base.control_model.ObsState* enum.

class ska_tango_base.csp.obs.obs_state_model.**CspSubElementObsStateModel**(*logger*, *callback=None*)

Implements the observation state model for a generic CSP sub-element ObsDevice.

Compared to the SKA observation state model, it implements a smaller number of states, number that can be further decreased depending on the necessities of the different sub-elements.

The implemented states are:

- **IDLE**: the device is unconfigured.
- **CONFIGURING**: transitional state to report device configuration is in progress.

  TODO: Need to understand if this state is really required by the observing devices of any CSP sub-element.
- **READY**: the device is configured and is ready to perform observations
- **SCANNING**: the device is performing the observation.
- **ABORTING**: the device is processing an abort.

  TODO: Need to understand if this state is really required by the observing devices of any CSP sub-element.
- **ABORTED**: the device has completed the abort request.
- **RESETTING**: the device is resetting from an ABORTED or FAULT state back to IDLE
- **FAULT**: the device component has experienced an error from which it can be recovered only via manual intervention invoking a reset command that force the device to the base state (IDLE).

A diagram of the CSP subelement observation state model is shown below. This model is non-deterministic as diagrammed, but the underlying state machine has extra state and transitions that render it deterministic. This model class simply maps those extra classes onto valid ObsState values

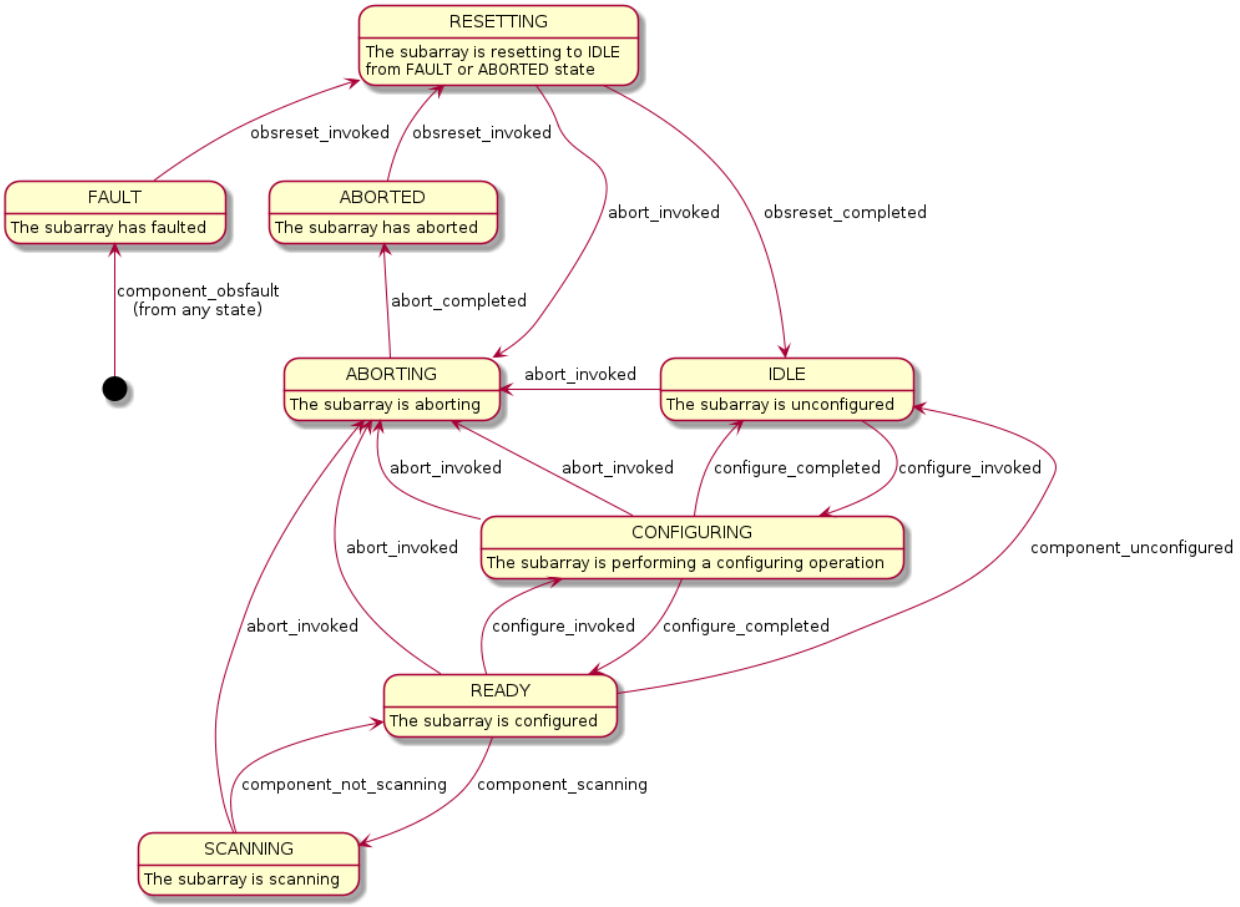


Fig. 4: Diagram of the observation state model

CSP obs component manager

This module models component management for CSP subelement observation devices.

class ska_tango_base.csp.obs.component_manager.**CspObsComponentManager**(*op_state_model*, *obs_state_model*, **args*, ***kwargs*)

A component manager for SKA CSP subelement observation 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.

configure_scan(*configuration*)

Configure the component.

Parameters

configuration (*dict*) – the configuration to be configured

deconfigure()

Deconfigure this component.

scan(*args*)

Start scanning.

end_scan()

End scanning.

abort()

Tell the component to abort whatever it was doing.

obsreset()

Reset the configuration but do not release resources.

property scan_id

Return the scan id.

property config_id

Return the configuration id.

component_configured(*configured*)

Handle notification that the component has started or stopped configuring.

This is callback hook.

Parameters
configured (*bool*) – whether this component is configured

component_scanning(*scanning*)

Handle notification that the component has started or stopped scanning.

This is a callback hook.

Parameters
scanning (*bool*) – whether this component is scanning

component_obsfault()

Handle notification that the component has obsfaulted.

This is a callback hook.

Reference CSP Obs Component Manager

This module models component management for CSP subelement observation devices.

ska_tango_base.csp.obs.reference_component_manager.**check_on**(*func*)

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 ska_tango_base.csp.obs.reference_component_manager.**ReferenceCspObsComponentManager**(*op_state_model*, *obs_state_model*, *logger=None*, *_component=None*)

A component manager for SKA CSP subelement observation 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.

start_communicating()
Establish communication with the component, then start monitoring.

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

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

Parameters
fail_communicate – whether the connection to the component is failing

configure_scan(*configuration*)
Configure the component.

deconfigure()
Tell the component to deconfigure.

scan(*args*)
Tell the component to start scanning.

end_scan()
Tell the component to stop scanning.

abort()
Cause the component to abort what it is doing.

obsreset()
Perform an obsreset on the component.

property scan_id
Return the scan id.

property config_id
Return the configuration id.

component_configured(*configured*)
Handle notification that the component has started or stopped configuring.

This is callback hook.

Parameters
configured (*bool*) – whether this component is configured

component_scanning(*scanning*)

Handle notification that the component has started or stopped scanning.

This is a callback hook.

Parameters
scanning (*bool*) – whether this component is scanning

component_obsfault()

Handle notification that the component has obsfaulted.

This is a callback hook.

CSP Subelement Obs device

CspSubElementObsDevice.

General observing device for SKA CSP Subelement.

class ska_tango_base.csp.obs.obs_device.**CspSubElementObsDevice**(**args: Any*, ***kwargs: Any*)

General observing device for SKA CSP Subelement.

Properties:

- **Device Property**

  DeviceID

  - Identification number of the observing device.
  - Type:'DevUShort'

scanID = tango.server.attribute(dtype=DevULong64, label=scanID, doc=The scan identification number to be inserted in the output products.)

Device attribute.

configurationID = tango.server.attribute(dtype=DevString, label=configurationID, doc=The configuration ID specified into the JSON configuration.)

Device attribute.

deviceID = tango.server.attribute(dtype=DevUShort, label=deviceID, doc=The observing device ID.)

Device attribute.

lastScanConfiguration = tango.server.attribute(dtype=DevString, label=lastScanConfiguration, doc=The last valid scan configuration.)

Device attribute.

sdpDestinationAddresses = tango.server.attribute(dtype=DevString, label=sdpDestinationAddresses, doc=JSON formatted string Report the list of all the SDP addresses provided by SDP to receive the output products. Specifies the Mac, IP, Port for each resource: CBF: visibility channels PSS ? Pss pipelines PST ? PSTBeam Not used by al CSP Sub-element observing device (for ex. Mid CBF VCCs))

Device attribute.

sdpLinkCapacity = tango.server.attribute(dtype=DevFloat, label=sdpLinkCapacity, doc=The SDP link capavity in GB/s.)

Device attribute.

sdpLinkActive = tango.server.attribute(dtype=('DevBoolean',), max_dim_x=100, label=sdpLinkActive, doc=Flag reporting if the SDP link is active. True: active False:down)

Device attribute.

healthFailureMessage = tango.server.attribute(dtype=DevString, label=healthFailureMessage, doc=Message providing info about device health failure.)

Device attribute.

create_component_manager()

Create and return the component manager for this device.

init_command_objects()

Set up the command objects.

class InitCommand(*target*, *op_state_model*, *logger=None*)

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

do()

Stateless hook for device initialisation.

Returns

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

Return type

(*ResultCode*, str)

always_executed_hook()

Perform actions before any Tango command is executed.

This is a Tango hook.

delete_device()

Clean up any resources prior to device deletion.

This method is a Tango hook that is called by the device destructor and by the device Init command. It allows for any memory or other resources allocated in the init_device method to be released prior to device deletion.

read_scanID()

Return the scanID attribute.

read_configurationID()

Return the configurationID attribute.

read_deviceID()

Return the deviceID attribute.

read_lastScanConfiguration()

Return the lastScanConfiguration attribute.

read_sdpDestinationAddresses()

Return the sdpDestinationAddresses attribute.

read_sdpLinkCapacity()

Return the sdpLinkCapacity attribute.

read_sdpLinkActive()

Return the sdpLinkActive attribute.

read_healthFailureMessage()

Return the healthFailureMessage attribute.

class ConfigureScanCommand(*target*, *op_state_model*, *obs_state_model*, *logger=None*)

A class for the CspSubElementObsDevices's ConfigureScan command.

do(*argin*)

Stateless hook for ConfigureScan() command functionality.

Parameters
argin (*dict*) – The configuration

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

Return type
(*ResultCode*, str)

validate_input(*argin*)

Validate the configuration parameters against allowed values, as needed.

Parameters
argin ('DevString') – The JSON formatted string with configuration for the device.

Returns
A tuple containing a return code and a string message.

Return type
(*ResultCode*, str)

class ScanCommand(*target*, *op_state_model*, *obs_state_model*, *logger=None*)

A class for the CspSubElementObsDevices's Scan command.

do(*argin*)

Stateless hook for Scan() command functionality.

Parameters
argin (*str*) – The scan ID.

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

Return type
(*ResultCode*, str)

validate_input(*argin*)

Validate the command input argument.

Parameters
argin (*string*) – the scan id

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

Return type
(*ResultCode*, str)

class EndScanCommand(*target*, *op_state_model*, *obs_state_model*, *logger=None*)

A class for the CspSubElementObsDevices's EndScan command.

do()

Stateless hook for EndScan() command functionality.

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

Return type
(*ResultCode*, str)

class GoToIdleCommand(*target*, *op_state_model*, *obs_state_model*, *logger=None*)

A class for the CspSubElementObsDevices's GoToIdle command.

do()

Stateless hook for GoToIdle() command functionality.

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

Return type
(*ResultCode*, str)

class ObsResetCommand(*target*, *op_state_model*, *obs_state_model*, *logger=None*)

A class for the CspSubElementObsDevices's ObsReset command.

do()

Stateless hook for ObsReset() command functionality.

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

Return type
(*ResultCode*, str)

class AbortCommand(*target*, *op_state_model*, *obs_state_model*, *logger=None*)

A class for the CspSubElementObsDevices's Abort command.

do()

Stateless hook for Abort() command functionality.

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

Return type
(*ResultCode*, str)

ConfigureScan(*argin*)

Configure the observing device parameters for the current scan.

Parameters
argin (*'DevString'*) – JSON formatted string with the scan configuration.

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

Return type
(*ResultCode*, str)

Scan(*argin*)

Start an observing scan.

Parameters
argin (*'DevString'*) – A string with the scan ID

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

Return type
(*ResultCode*, str)

EndScan()

End a running scan.

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

Return type
(*ResultCode*, str)

GoToIdle()

Transit the device from READY to IDLE obsState.

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

Return type
(*ResultCode*, str)

ObsReset()

Reset the observing device from a FAULT/ABORTED obsState to IDLE.

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

Return type
(*ResultCode*, str)

Abort()

Abort the current observing process and move the device to ABORTED obsState.

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

Return type
(*ResultCode*, str)

ska_tango_base.csp.obs.obs_device.**main**(*args=None*, ***kwargs*)

Run the CspSubElementObsDevice module.

2.3.2 CSP subarray subpackage

This subpackage contains subarray device functionality specific to CSP.

CSP Subarray Component Manager

This module models component management for CSP subarrays.

class ska_tango_base.csp.subarray.component_manager.**CspSubarrayComponentManager**(*op_state_model*, *obs_state_model*, **args*, ***kwargs*)

A component manager for SKA CSP 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.

property config_id

Return the configuration id.

property scan_id

Return the scan id.

Reference CSP Subarray Component Manager

This module models component management for CSP subelement observation devices.

ska_tango_base.csp.subarray.reference_component_manager.**check_on**(*func*)

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 ska_tango_base.csp.subarray.reference_component_manager.**ReferenceCspSubarrayComponentManager**(*op_st*
obs_s
ca-
pa-
bil-
ity_ty
log-
ger,
_com
po-
nent=

A component manager for SKA CSP subelement observation 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.

property config_id

Return the configuration id.

property scan_id

Return the scan id.

CSP Subarray device

CspSubElementSubarray.

Subarray device for SKA CSP SubElement

class ska_tango_base.csp.subarray.subarray_device.**CspSubElementSubarray**(**args: Any*, ***kwargs: Any*)

Subarray device for SKA CSP SubElement.

scanID = tango.server.attribute(dtype=DevULong64, label=scanID, doc=The scan identification number to be inserted in the output products.)

Device attribute.

configurationID = tango.server.attribute(dtype=DevString, label=configurationID, doc=The configuration ID specified into the JSON configuration.)

Device attribute.

sdpDestinationAddresses = tango.server.attribute(dtype=DevString, access=tango.AttrWriteType.READ_WRITE, label=sdpDestinationAddresses, doc=JSON formatted string. Report the list of all the SDP addresses provided by SDP to receive the output products. Specifies the Mac, IP, Port for each resource:CBF visibility channels, Pss pipelines, PSTBeam)

Device attribute.

outputDataRateToSdp = tango.server.attribute(dtype=DevFloat, label=outputDataRateToSdp, unit=GB/s, doc=The output data rate (GB/s) on the link for each scan.)

Device attribute.

lastScanConfiguration = tango.server.attribute(dtype=DevString, label=lastScanConfiguration, doc=The last valid scan configuration.)

Device attribute.

sdpLinkActive = tango.server.attribute(dtype=('DevBoolean',), max_dim_x=100, label=sdpLinkActive, doc=Flag reporting if the SDP links are active.)

Device attribute.

listOfDevicesCompletedTasks = tango.server.attribute(dtype=DevString, label=listOfDevicesCompletedTasks, doc=JSON formatted string reporting for each task/command the list of devices that completed successfully the task. Ex. {``cmd1``: [``device1``, ``device2``], ``cmd2``: [``device2``, ``device3``]})

Device attribute.

configureScanMeasuredDuration = tango.server.attribute(dtype=DevFloat, label=configureScanMeasuredDuration, unit=sec, doc=The measured time (sec) taken to execute the command)

Device attribute.

configureScanTimeoutExpiredFlag = tango.server.attribute(dtype=DevBoolean, label=configureScanTimeoutExpiredFlag, doc=Flag reporting ConfigureScan command timeout expiration.)

Device attribute.

assignResourcesMaximumDuration = tango.server.attribute(dtype=DevFloat, access=tango.AttrWriteType.READ_WRITE, label=assignResourcesMaximumDuration, unit=sec, doc=The maximum expected command duration.)

Device attribute.

assignResourcesMeasuredDuration = tango.server.attribute(dtype=DevFloat, label=assignResourcesMeasuredDuration, unit=sec, doc=The measured command execution duration.)

Device attribute.

assignResourcesProgress = tango.server.attribute(dtype=DevUShort, label=assignResourcesProgress, max_value=100, min_value=0, doc=The percentage progress of the command in the [0,100].)

Device attribute.

assignResourcesTimeoutExpiredFlag = tango.server.attribute(dtype=DevBoolean, label=assignResourcesTimeoutExpiredFlag, doc=Flag reporting AssignResources command timeout expiration.)

Device attribute.

releaseResourcesMaximumDuration = tango.server.attribute(dtype=DevFloat, access=tango.AttrWriteType.READ_WRITE, label=releaseResourcesMaximumDuration, unit=sec, doc=The maximum expected command duration.)

Device attribute.

releaseResourcesMeasuredDuration = tango.server.attribute(dtype=DevFloat, label=releaseResourcesMeasuredDuration, unit=sec, doc=The measured command execution duration.)

Device attribute.

releaseResourcesProgress = tango.server.attribute(dtype=DevUShort, label=releaseResourcesProgress, max_value=100, min_value=0, doc=The percentage progress of the command in the [0,100].)

Device attribute.

releaseResourcesTimeoutExpiredFlag = tango.server.attribute(dtype=DevBoolean, label=timeoutExpiredFlag, doc=Flag reporting command timeout expiration.)

Device attribute.

create_component_manager()

Create and return the component manager for this device.

init_command_objects()

Set up the command objects.

class InitCommand(*target*, *op_state_model*, *logger=None*)

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

do()

Stateless hook for device initialisation.

Returns

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

Return type

(*ResultCode*, str)

always_executed_hook()

Perform actions that are executed before every device command.

This is a Tango hook.

delete_device()

Clean up any resources prior to device deletion.

This method is a Tango hook that is called by the device destructor and by the device Init command. It allows for any memory or other resources allocated in the init_device method to be released prior to device deletion.

read_scanID()

Return the scanID attribute.

read_configurationID()

Return the configurationID attribute.

read_sdpDestinationAddresses()

Return the sdpDestinationAddresses attribute.

write_sdpDestinationAddresses(*value*)

Set the sdpDestinationAddresses attribute.

read_outputDataRateToSdp()

Return the outputDataRateToSdp attribute.

read_lastScanConfiguration()

Return the lastScanConfiguration attribute.

read_configureScanMeasuredDuration()

Return the configureScanMeasuredDuration attribute.

read_configureScanTimeoutExpiredFlag()

Return the configureScanTimeoutExpiredFlag attribute.

read_listOfDevicesCompletedTasks()

Return the listOfDevicesCompletedTasks attribute.

read_assignResourcesMaximumDuration()

Return the assignResourcesMaximumDuration attribute.

write_assignResourcesMaximumDuration(*value*)

Set the assignResourcesMaximumDuration attribute.

read_assignResourcesMeasuredDuration()

Return the assignResourcesMeasuredDuration attribute.

read_assignResourcesProgress()

Return the assignResourcesProgress attribute.

read_assignResourcesTimeoutExpiredFlag()

Return the assignResourcesTimeoutExpiredFlag attribute.

read_releaseResourcesMaximumDuration()

Return the releaseResourcesMaximumDuration attribute.

write_releaseResourcesMaximumDuration(*value*)

Set the releaseResourcesMaximumDuration attribute.

read_releaseResourcesMeasuredDuration()

Return the releaseResourcesMeasuredDuration attribute.

read_releaseResourcesProgress()

Return the releaseResourcesProgress attribute.

read_releaseResourcesTimeoutExpiredFlag()

Return the releaseResourcesTimeoutExpiredFlag attribute.

read_sdpLinkActive()

Return the sdpLinkActive attribute.

class ConfigureScanCommand(*target*, *op_state_model*, *obs_state_model*, *logger=None*)

A class for the CspSubElementObsDevices's ConfigureScan command.

do(*argin*)

Stateless hook for ConfigureScan() command functionality.

Parameters
argin (*dict*) – The configuration

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

Return type
(*ResultCode*, str)

validate_input(*argin*)

Validate the configuration parameters against allowed values, as needed.

Parameters
argin (*'DevString'*) – The JSON formatted string with configuration for the device.

Returns
A tuple containing a return code and a string message.

Return type
(*ResultCode*, str)

class GoToIdleCommand(*target*, *op_state_model*, *obs_state_model*, *logger=None*)

A class for the CspSubElementObsDevices's GoToIdle command.

do()

Stateless hook for GoToIdle() command functionality.

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

Return type
(*ResultCode*, str)

ConfigureScan(*argin*)

Configure a complete scan for the subarray.

Parameters
argin (*'DevString'*) – JSON formatted string with the scan configuration.

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

Return type
(*ResultCode*, str)

Configure(*argin*)

Redirect to ConfigureScan method. Configure a complete scan for the subarray.

Parameters
argin – JSON configuration string

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

GoToIdle()

Transit the subarray from READY to IDLE obsState.

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

End()

Transit the subarray from READY to IDLE obsState. Redirect to GoToIdle command.

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

ska_tango_base.csp.subarray.subarray_device.**main**(*args=None*, ***kwargs*)

Run the CspSubElementSubarray module.

2.3.3 CSP Subelement Controller device

CspSubElementController.

Controller device for SKA CSP Subelement.

class ska_tango_base.csp.controller_device.**CspSubElementController**(**args: Any*, ***kwargs: Any*)

Controller device for SKA CSP Subelement.

Properties:

- **Device Property**

  PowerDelayStandbyOn

  - Delay in sec between power-up stages in Standby<-> On transitions.
  - Type:'DevFloat'

  PowerDelayStandByOff

  - Delay in sec between power-up stages in Standby-> Off transition.
  - Type:'DevFloat'

powerDelayStandbyOn = tango.server.attribute(dtype=DevFloat, access=tango.AttrWriteType.READ_WRITE, label=powerDelayStandbyOn, unit=sec., doc=Delay in sec between the power-up stages in Standby<->On transitions.)

Device attribute.

powerDelayStandbyOff = tango.server.attribute(dtype=DevFloat, access=tango.AttrWriteType.READ_WRITE, label=powerDelayStandbyOff, unit=sec, doc=Delay in sec between the power-up stages in Standby->Off transitions.)

Device attribute.

onProgress = tango.server.attribute(dtype=DevUShort, label=onProgress, max_value=100, min_value=0, doc=Progress percentage of the command execution.)

Device attribute.

onMaximumDuration = tango.server.attribute(dtype=DevFloat, access=tango.AttrWriteType.READ_WRITE, label=onMaximumDuration, unit=sec., doc=The expected maximum duration (sec.) to execute the On command.)

Device attribute.

onMeasuredDuration = tango.server.attribute(dtype=DevFloat, label=onMeasuredDuration, unit=sec, doc=The measured time (sec) taken to execute the command.)

Device attribute.

standbyProgress = tango.server.attribute(dtype=DevUShort, label=standbyProgress, max_value=100, min_value=0, doc=Progress percentage of the command execution.)

Device attribute.

standbyMaximumDuration = tango.server.attribute(dtype=DevFloat, access=tango.AttrWriteType.READ_WRITE, label=standbyMaximumDuration, unit=sec., doc=The expected maximum duration (sec.) to execute the Standby command.)

Device attribute.

standbyMeasuredDuration = tango.server.attribute(dtype=DevFloat, label=standbyMeasuredDuration, unit=sec, doc=The measured time (sec) taken to execute the Standby command.)

Device attribute.

offProgress = tango.server.attribute(dtype=DevUShort, label=offProgress, max_value=100, min_value=0, doc=Progress percentage of the command execution.)

Device attribute.

offMaximumDuration = tango.server.attribute(dtype=DevFloat, access=tango.AttrWriteType.READ_WRITE, label=offMaximumDuration, unit=sec., doc=The expected maximum duration (sec.) to execute the Off command.)

Device attribute.

offMeasuredDuration = tango.server.attribute(dtype=DevFloat, label=offMeasuredDuration, unit=sec, doc=The measured time (sec) taken to execute the Off command.)

Device attribute.

totalOutputDataRateToSdp = tango.server.attribute(dtype=DevFloat, label=totalOutputDataRateToSdp, unit=GB/s, doc=Report the total link expected output data rate.)

Device attribute.

loadFirmwareProgress = tango.server.attribute(dtype=DevUShort, label=loadFirmwareProgress, max_value=100, min_value=0, doc=The command progress percentage.)

Device attribute.

loadFirmwareMaximumDuration = tango.server.attribute(dtype=DevFloat, access=tango.AttrWriteType.READ_WRITE, label=loadFirmwareMaximumDuration, unit=sec, doc=The expected maximum duration (in sec) for command execution.)

Device attribute.

loadFirmwareMeasuredDuration = tango.server.attribute(dtype=DevFloat, label=loadFirmwareMeasuredDuration, unit=sec, doc=The command execution measured duration (in sec).)

Device attribute.

init_command_objects()

Set up the command objects.

class InitCommand(*target*, *op_state_model*, *logger=None*)

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

do()

Stateless hook for device initialisation.

Returns

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

Return type

(*ResultCode*, str)

always_executed_hook()

Perform actions always executed before any Tango command is executed.

This is a Tango hook.

delete_device()

Clean up any resources prior to device deletion.

This method is a Tango hook that is called by the device destructor and by the device Init command. It allows for any memory or other resources allocated in the init_device method to be released prior to device deletion.

read_powerDelayStandbyOn()

Return the powerDelayStandbyOn attribute.

write_powerDelayStandbyOn(*value*)

Set the powerDelayStandbyOn attribute.

read_onProgress()

Return the onProgress attribute.

read_onMaximumDuration()

Return the onMaximumDuration attribute.

write_onMaximumDuration(*value*)

Set the onMaximumDuration attribute.

read_onMeasuredDuration()

Return the onMeasuredDuration attribute.

read_standbyProgress()
Return the standbyProgress attribute.

read_standbyMaximumDuration()
Return the standbyMaximumDuration attribute.

write_standbyMaximumDuration(*value*)
Set the standbyMaximumDuration attribute.

read_standbyMeasuredDuration()
Return the standbyMeasuredDuration attribute.

read_offProgress()
Return the offProgress attribute.

read_offMaximumDuration()
Return the offMaximumDuration attribute.

write_offMaximumDuration(*value*)
Set the offMaximumDuration attribute.

read_offMeasuredDuration()
Return the offMeasuredDuration attribute.

read_totalOutputDataRateToSdp()
Return the totalOutputDataRateToSdp attribute.

read_powerDelayStandbyOff()
Return the powerDelayStandbyOff attribute.

write_powerDelayStandbyOff(*value*)
Set the powerDelayStandbyOff attribute.

read_loadFirmwareProgress()
Return the loadFirmwareProgress attribute.

read_loadFirmwareMaximumDuration()
Return the loadFirmwareMaximumDuration attribute.

write_loadFirmwareMaximumDuration(*value*)
Set the loadFirmwareMaximumDuration attribute.

read_loadFirmwareMeasuredDuration()
Return the loadFirmwareMeasuredDuration attribute.

class LoadFirmwareCommand(*target*, *op_state_model*, *admin_mode_model*, **args*, *logger=None*, ***kwargs*)
A class for the LoadFirmware command.

do(*argin*)
Stateless hook for device LoadFirmware() command.

Parameters
argin – argument to command, currently unused

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

Return type
(*ResultCode*, str)

is_allowed(*raise_if_disallowed=False*)

Check if the command is in the proper state to be executed.

The controller device has to be in op state OFF and admin mode MAINTENACE to process the Load-Firmware command.

Parameters
raise_if_disallowed – whether to raise an error or simply return False if the command is disallowed

Raises
CommandError – if command not allowed

Returns
`True` if the command is allowed.

Return type
boolean

class PowerOnDevicesCommand(*target*, *op_state_model*, **args*, *logger=None*, ***kwargs*)

A class for the CspSubElementController's PowerOnDevices command.

do(*argin*)

Stateless hook for device PowerOnDevices() command.

Parameters
argin – argument to command, currently unused

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

Return type
(*ResultCode*, str)

is_allowed(*raise_if_disallowed=False*)

Check if the command is in the proper state to be executed.

The controller device has to be in ON to process the PowerOnDevices command.

Parameters
raise_if_disallowed – whether to raise an error or simply return False if the command is disallowed

Raises
CommandError – if command not allowed

Returns
`True` if the command is allowed.

Return type
boolean

class PowerOffDevicesCommand(*target*, *op_state_model*, **args*, *logger=None*, ***kwargs*)

A class for the CspSubElementController's PowerOffDevices command.

do(*argin*)

Stateless hook for device PowerOffDevices() command.

Parameters
argin – argument to command, currently unused

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

Return type
(*ResultCode*, str)

is_allowed(*raise_if_disallowed=False*)

Check if the command is in the proper state to be executed.

The controller device has to be in ON to process the PowerOffDevices command.

Parameters
raise_if_disallowed – whether to raise an error or simply return False if the command is disallowed

Raises
CommandError – if command not allowed

Returns
`True` if the command is allowed.

Return type
boolean

class ReInitDevicesCommand(*target*, *op_state_model*, **args*, *logger=None*, ***kwargs*)

A class for the CspSubElementController's ReInitDevices command.

do(*argin*)

Stateless hook for device ReInitDevices() command.

Parameters
argin – argument to command, currently unused

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

Return type
(*ResultCode*, str)

is_allowed(*raise_if_disallowed=False*)

Check if the command is in the proper state to be executed.

The controller device has to be in ON to process the ReInitDevices command.

Parameters
raise_if_disallowed – whether to raise an error or simply return False if the command is disallowed

Raises
CommandError – if command not allowed

Returns
`True` if the command is allowed.

Return type
boolean

is_LoadFirmware_allowed()

Check if the LoadFirmware command is allowed in the current state.

Returns
`True` if command is allowed

Return type
boolean

LoadFirmware(*argin*)

Deploy new versions of software and firmware.

After deployment, a restart is triggers so that a Component initializes using a newly deployed version.

Parameters
argin (*'DevVarStringArray'*) – A list of three strings: - The file name or a pointer to

the filename specified as URL. - the list of components that use software or firmware package (file), - checksum or signing Ex: ['file://firmware.txt','test/dev/1, test/dev/2, test/dev/3', '918698a7fea3fa9da5996db001d33628']

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

Return type
(*ResultCode*, str)

`is_PowerOnDevices_allowed()`

Check if the PowerOnDevice command is allowed in the current state.

Returns
`True` if command is allowed

Return type
boolean

`PowerOnDevices(argin)`

Power-on a selected list of devices.

Parameters
argin (*'DevVarStringArray'*) – List of devices (FQDNs) to power-on.

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

Return type
(*ResultCode*, str)

`is_PowerOffDevices_allowed()`

Check if the PowerOffDevices command is allowed in the current state.

Returns
`True` if command is allowed

Return type
boolean

`PowerOffDevices(argin)`

Power-off a selected list of devices.

Parameters
argin (*'DevVarStringArray'*) – List of devices (FQDNs) to power-off.

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

Return type
(*ResultCode*, str)

`is_ReInitDevices_allowed()`

Check if the ReInitDevices command is allowed in the current state.

Returns
`True` if command is allowed

Return type
boolean

ReInitDevices(*argin*)

Reinitialize the devices passed in the input argument.

The exact functionality may vary for different devices and sub-systems, each Tango Device/Server should define what does ReInitDevices means. Ex: ReInitDevices FPGA -> reset ReInitDevices Controller -> Restart ReInitDevices Leaf PC -> reboot

Parameters
argin (*'DevVarStringArray'*) – List of devices (FQDNs) to re-initialize.

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

Return type
(*ResultCode*, str)

ska_tango_base.csp.controller_device.**main**(*args=None*, ***kwargs*)

Entry point for the CspSubElementController module.

Parameters

- **args** – str
- **kwargs** – str

Returns
exit code

2.4 Subarray

This subpackage models a SKA subarray Tango device.

2.4.1 Subarray Obs State Model

This module specifies the observation state model for SKA subarray Tango devices.

It consists of:

- an underlying state machine: _SubarrayObsStateMachine
- an *SubarrayObsStateModel* that maps the underlying state machine state to a value of the *ska_tango_base.control_model.ObsState* enum.

class ska_tango_base.subarray.subarray_obs_state_model.**SubarrayObsStateModel**(*logger*, *callback=None*)

Implements the observation state model for subarray.

The model supports all of the states of the *ska_tango_base.control_model.ObsState* enum:

- **EMPTY**: the subarray is unresourced
- **RESOURCING**: the subarray is performing a resourcing operation
- **IDLE**: the subarray is resourced but unconfigured
- **CONFIGURING**: the subarray is performing a configuring operation

- **READY**: the subarray is resourced and configured
- **SCANNING**: the subarray is scanning
- **ABORTING**: the subarray is aborting
- **ABORTED**: the subarray has aborted
- **RESETTING**: the subarray is resetting from an ABORTED or FAULT state back to IDLE
- **RESTARTING**: the subarray is restarting from an ABORTED or FAULT state back to EMPTY
- **FAULT**: the subarray has encountered a observation fault.

A diagram of the subarray observation state model is shown below. This model is non-deterministic as diagrammed, but the underlying state machines has extra states and transitions that render it deterministic. This class simply maps those extra classes onto valid ObsState values.

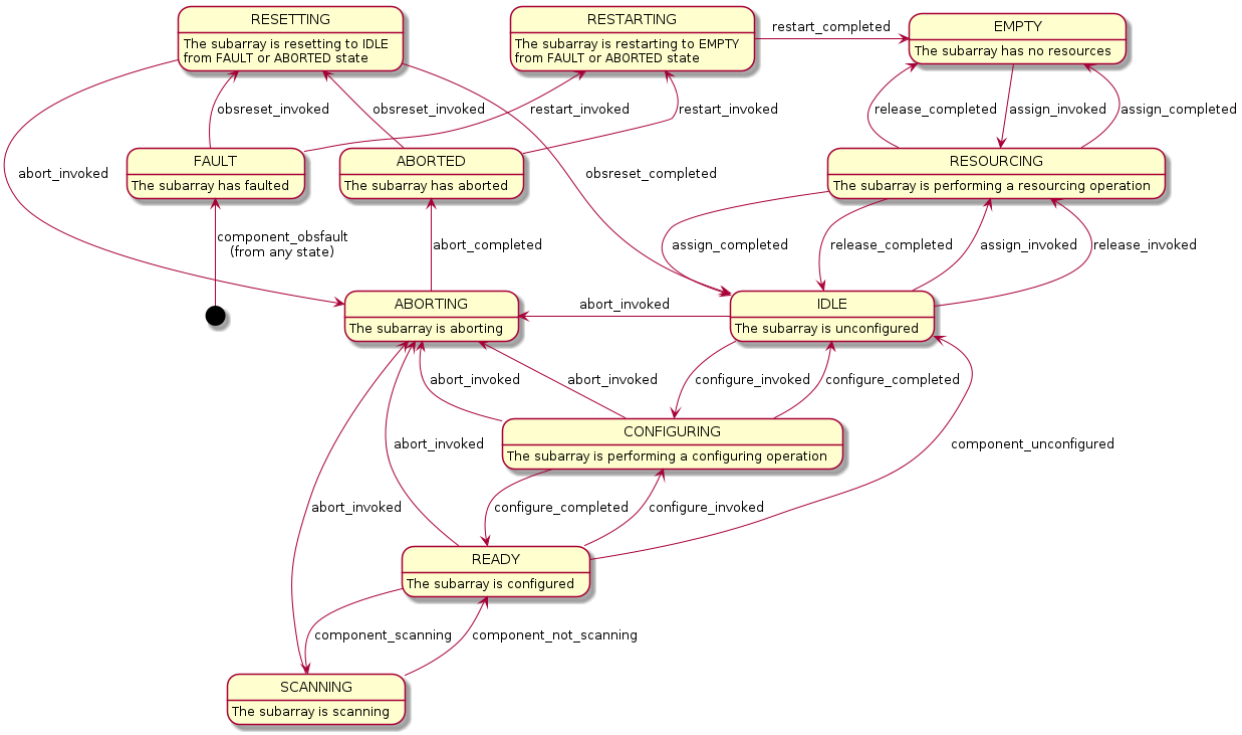


Fig. 5: Diagram of the subarray observation state model

2.4.2 Subarray Component Manager

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

class ska_tango_base.subarray.component_manager.**SubarrayComponentManager**(*op_state_model*, *obs_state_model*)

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*)

Assign resources to the component.

Parameters
resources – resources to be assigned

release(*resources*)

Release resources from the component.

Parameters
resources – resources to be released

release_all()

Release all resources.

configure(*configuration*)

Configure the component.

Parameters
configuration (*dict*) – the configuration to be configured

deconfigure()

Deconfigure this component.

scan(*args*)

Start scanning.

end_scan()

End scanning.

abort()

Tell the component to abort whatever it was doing.

obsreset()

Reset the component to unconfigured but do not release resources.

restart()

Deconfigure and release all resources.

property assigned_resources

Return the resources assigned to the component.

Returns
the resources assigned to the component

Return type
list of str

property configured_capabilities

Return the configured capabilities of the component.

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

Return type
list of str

component_resourced(*resourced*)

Handle notification that the component's resources have changed.

This is a callback hook.

Parameters
resourced (*bool*) – whether this component has any resources

component_configured(*configured*)

Handle notification that the component has started or stopped configuring.

This is callback hook.

Parameters
configured (*bool*) – whether this component is configured

component_scanning(*scanning*)

Handle notification that the component has started or stopped scanning.

This is a callback hook.

Parameters
scanning (*bool*) – whether this component is scanning

component_obsfault()

Handle notification that the component has obsfaulted.

This is a callback hook.

2.4.3 Reference Subarray Component Manager

This module models component management for SKA subarray devices.

ska_tango_base.subarray.reference_component_manager.**check_on**(*func*)

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 ska_tango_base.subarray.reference_component_manager.**ReferenceSubarrayComponentManager**(*op_state_mode*, *obs_state_mod*, *capability_types*, *logger=None*, *_component=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.

start_communicating()

Establish communication with the component, then start monitoring.

stop_communicating()

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

simulate_communication_failure(*fail_communicate*)

Simulate (or stop simulating) a component connection failure.

Parameters
fail_communicate – whether the connection to the component is failing

assign(*resources*)

Assign resources to the component.

Parameters
resources (*list(str)*) – resources to be assigned

release(*resources*)

Release resources from the component.

Parameters
resources (*list(str)*) – resources to be released

release_all()

Release all resources.

configure(*configuration*)

Configure the component.

Parameters
configuration (*dict*) – the configuration to be configured

deconfigure()

Deconfigure this component.

scan(*args*)

Start scanning.

end_scan()

End scanning.

abort()

Tell the component to abort whatever it was doing.

obsreset()

Deconfigure the component but do not release resources.

restart()

Tell the component to restart.

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

property assigned_resources

Return the resources assigned to the component.

Returns
the resources assigned to the component

Return type
list of str

property configured_capabilities

Return the configured capabilities of the component.

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

Return type
list of str

component_resourced(*resourced*)

Handle notification that the component's resources have changed.

This is a callback hook.

Parameters
resourced (*bool*) – whether this component has any resources

component_configured(*configured*)

Handle notification that the component has started or stopped configuring.

This is a callback hook.

Parameters
configured (*bool*) – whether this component is configured

component_scanning(*scanning*)

Handle notification that the component has started or stopped scanning.

This is a callback hook.

Parameters
scanning (*bool*) – whether this component is scanning

component_obsfault()

Handle notification that the component has obsfaulted.

This is a callback hook.

2.4.4 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 ska_tango_base.subarray.subarray_device.**SKASubarray**(**args: Any*, ***kwargs: Any*)

Implements the SKA SubArray device.

class InitCommand(*target*, *op_state_model*, *logger=None*)

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

do()

Stateless hook for device initialisation.

Returns

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

Return type

(*ResultCode*, str)

class AssignResourcesCommand(*target*, *op_state_model*, *obs_state_model*, *logger=None*)

A class for SKASubarray's AssignResources() command.

do(*argin*)

Stateless hook for AssignResources() command functionality.

Parameters

argin (*list of str*) – The resources to be assigned

Returns

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

Return type

(*ResultCode*, str)

class ReleaseResourcesCommand(*target*, *op_state_model*, *obs_state_model*, *logger=None*)

A class for SKASubarray's ReleaseResources() command.

do(*argin*)

Stateless hook for ReleaseResources() command functionality.

Parameters

argin (*list of str*) – The resources to be released

Returns

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

Return type

(*ResultCode*, str)

class ReleaseAllResourcesCommand(*target*, *op_state_model*, *obs_state_model*, *logger=None*)

A class for SKASubarray's ReleaseAllResources() command.

do()

Stateless hook for ReleaseAllResources() command functionality.

Returns

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

Return type

(*ResultCode*, str)

class ConfigureCommand(*target*, *op_state_model*, *obs_state_model*, *logger=None*)

A class for SKASubarray's Configure() command.

do(*argin*)

Stateless hook for Configure() command functionality.

Parameters
argin (*str*) – The configuration as JSON

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

Return type
(*ResultCode*, str)

class ScanCommand(*target*, *op_state_model*, *obs_state_model*, *logger=None*)

A class for SKASubarray's Scan() command.

do(*argin*)

Stateless hook for Scan() command functionality.

Parameters
argin (*str*) – Scan info

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

Return type
(*ResultCode*, str)

class EndScanCommand(*target*, *op_state_model*, *obs_state_model*, *logger=None*)

A class for SKASubarray's EndScan() command.

do()

Stateless hook for EndScan() command functionality.

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

Return type
(*ResultCode*, str)

class EndCommand(*target*, *op_state_model*, *obs_state_model*, *logger=None*)

A class for SKASubarray's End() command.

do()

Stateless hook for End() command functionality.

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

Return type
(*ResultCode*, str)

class AbortCommand(*target*, *op_state_model*, *obs_state_model*, *logger=None*)

A class for SKASubarray's Abort() command.

do()

Stateless hook for Abort() command functionality.

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

Return type
(*ResultCode*, str)

class ObsResetCommand(*target*, *op_state_model*, *obs_state_model*, *logger=None*)

A class for SKASubarray's ObsReset() command.

do()

Stateless hook for ObsReset() command functionality.

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

Return type
(*ResultCode*, str)

class RestartCommand(*target*, *op_state_model*, *obs_state_model*, *logger=None*)

A class for SKASubarray's Restart() command.

do()

Execute the functionality of the Restart() command.

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

Return type
(*ResultCode*, str)

create_component_manager()

Create and return a component manager for this device.

init_command_objects()

Set up the command objects.

activationTime = tango.server.attribute(dtype=double, unit=s, standard_unit=s, display_unit=s, doc=Time of activation in seconds since Unix epoch.)

Device attribute.

assignedResources = tango.server.attribute(dtype=('str',), max_dim_x=100, doc=The list of resources assigned to the subarray.)

Device attribute.

configuredCapabilities = tango.server.attribute(dtype=('str',), max_dim_x=10, doc=A list of capability types with no. of instances in use on this subarray; e.g. Correlators:512, PssBeams:4, PstBeams:4, VlbiBeams:0.)

Device attribute.

always_executed_hook()

Perform actions that are executed before every device command.

This is a Tango hook.

delete_device()

Clean up any resources prior to device deletion.

This method is a Tango hook that is called by the device destructor and by the device Init command. It allows for any memory or other resources allocated in the init_device method to be released prior to device deletion.

`read_activationTime()`

Read the time since device is activated.

Returns
Time of activation in seconds since Unix epoch.

`read_assignedResources()`

Read the resources assigned to the device.

Returns
Resources assigned to the device.

`read_configuredCapabilities()`

Read capabilities configured in the Subarray.

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

`is_AssignResources_allowed()`

Check if command *AssignResources* is allowed in the current device state.

Returns
`True` if the command is allowed

Return type
boolean

`AssignResources(argin)`

Assign resources to this subarray.

To modify behaviour for this command, modify the do() method of the command class.

Parameters
argin (*list of str*) – the resources to be assigned

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

Return type
(*ResultCode*, str)

`is_ReleaseResources_allowed()`

Check if command *ReleaseResources* is allowed in the current device state.

Returns
`True` if the command is allowed

Return type
boolean

`ReleaseResources(argin)`

Delta removal of assigned resources.

To modify behaviour for this command, modify the do() method of the command class.

Parameters
argin (*list of str*) – the resources to be released

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

Return type
(*ResultCode*, str)

`is_ReleaseAllResources_allowed()`

Check if command *ReleaseAllResources* is allowed in the current device state.

Returns
`True` if the command is allowed

Return type
boolean

`ReleaseAllResources()`

Remove all resources to tear down to an empty subarray.

To modify behaviour for this command, modify the do() method of the command class.

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

Return type
(*ResultCode*, str)

`is_Configure_allowed()`

Check if command *Configure* is allowed in the current device state.

Returns
`True` if the command is allowed

Return type
boolean

`Configure(argin)`

Configure the capabilities of this subarray.

To modify behaviour for this command, modify the do() method of the command class.

Parameters
argin (*string*) – configuration specification

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

Return type
(*ResultCode*, str)

`is_Scan_allowed()`

Check if command *Scan* is allowed in the current device state.

Returns
`True` if the command is allowed

Return type
boolean

`Scan(argin)`

Start scanning.

To modify behaviour for this command, modify the do() method of the command class.

Parameters
argin (*Array of str*) – Information about the scan

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

Return type
(*ResultCode*, str)

`is_EndScan_allowed()`

Check if command *EndScan* is allowed in the current device state.

Returns
`True` if the command is allowed

Return type
boolean

`EndScan()`

End the scan.

To modify behaviour for this command, modify the do() method of the command class.

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

Return type
(*ResultCode*, str)

`is_End_allowed()`

Check if command *End* is allowed in the current device state.

Returns
`True` if the command is allowed

Return type
boolean

`End()`

End the scan block.

To modify behaviour for this command, modify the do() method of the command class.

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

Return type
(*ResultCode*, str)

`is_Abort_allowed()`

Check if command *Abort* is allowed in the current device state.

Returns
`True` if the command is allowed

Return type
boolean

Abort()

Abort any long-running command such as `Configure()` or `Scan()`.

To modify behaviour for this command, modify the do() method of the command class.

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

Return type
(*ResultCode*, str)

is_ObsReset_allowed()

Check if command *ObsReset* is allowed in the current device state.

Returns
`True` if the command is allowed

Return type
boolean

ObsReset()

Reset the current observation process.

To modify behaviour for this command, modify the do() method of the command class.

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

Return type
(*ResultCode*, str)

is_Restart_allowed()

Check if command *Restart* is allowed in the current device state.

Returns
`True` if the command is allowed

Return type
boolean

Restart()

Restart the subarray. That is, deconfigure and release all resources.

To modify behaviour for this command, modify the do() method of the command class.

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

Return type
(*ResultCode*, str)

ska_tango_base.subarray.subarray_device.**main**(*args=None*, ***kwargs*)

Launch an SKASubarray device.

Parameters

- **args** – positional args to tango.server.run
- **kwargs** – named args to tango.server.run

Returns
exit code

2.5 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 ska_tango_base.alarm_handler_device.**SKAAlarmHandler**(**args: Any*, ***kwargs: Any*)

A generic base device for Alarms for SKA.

statsNrAlerts = tango.server.attribute(dtype=int, doc=Number of active Alerts)

Device attribute.

statsNrAlarms = tango.server.attribute(dtype=int, doc=Number of active Alarms)

Device attribute.

statsNrNewAlarms = tango.server.attribute(dtype=int, doc=Number of New active alarms)

Device attribute.

statsNrUnackAlarms = tango.server.attribute(dtype=double, doc=Number of unacknowledged alarms)

Device attribute.

statsNrRtnAlarms = tango.server.attribute(dtype=double, doc=Number of returned alarms)

Device attribute.

activeAlerts = tango.server.attribute(dtype=('str',), max_dim_x=10000, doc=List of active alerts)

Device attribute.

activeAlarms = tango.server.attribute(dtype=('str',), max_dim_x=10000, doc=List of active alarms)

Device attribute.

init_command_objects()

Set up the command objects.

always_executed_hook()

Perform actions that are executed before every device command.

This is a Tango hook.

delete_device()

Clean up any resources prior to device deletion.

This method is a Tango hook that is called by the device destructor and by the device Init command. It allows for any memory or other resources allocated in the init_device method to be released prior to device deletion.

read_statsNrAlerts()

Read number of active alerts.

Returns
Number of active alerts

read_statsNrAlarms()

Read number of active alarms.

Returns
Number of active alarms

read_statsNrNewAlarms()

Read number of new active alarms.

Returns
Number of new active alarms

read_statsNrUnackAlarms()

Read number of unacknowledged alarms.

Returns
Number of unacknowledged alarms.

read_statsNrRtnAlarms()

Read number of returned alarms.

Returns
Number of returned alarms

read_activeAlerts()

Read list of active alerts.

Returns
List of active alerts

read_activeAlarms()

Read list of active alarms.

Returns
List of active alarms

class GetAlarmRuleCommand(*target*, **args*, *logger=None*, ***kwargs*)

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

do(*argin*)

Stateless hook for SKAAlarmHandler GetAlarmRule() command.

Returns
Alarm configuration info: rule, actions, etc.

Return type
JSON string

class GetAlarmDataCommand(*target*, **args*, *logger=None*, ***kwargs*)

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

do(*argin*)

Stateless hook for SKAAlarmHandler GetAlarmData() command.

Returns
Alarm data

Return type
JSON string

class GetAlarmAdditionalInfoCommand(*target*, **args*, *logger=None*, ***kwargs*)

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

do(*argin*)

Stateless hook for SKAAlarmHandler GetAlarmAdditionalInfo() command.

Returns
Alarm additional info
Return type
JSON string

class GetAlarmStatsCommand(*target*, **args*, *logger=None*, ***kwargs*)

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

do()

Stateless hook for SKAAlarmHandler GetAlarmStats() command.

Returns
Alarm stats
Return type
JSON string

class GetAlertStatsCommand(*target*, **args*, *logger=None*, ***kwargs*)

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

do()

Stateless hook for SKAAlarmHandler GetAlertStats() command.

Returns
Alert stats
Return type
JSON string

GetAlarmRule(*argin*)

Get all configuration info of the alarm, e.g. rule, defined action, etc.

To modify behaviour for this command, modify the do() method of the command class.

Parameters
argin – Name of the alarm

Returns
JSON string containing configuration information of the alarm

GetAlarmData(*argin*)

Get data on all attributes participating in the alarm rule.

The data includes current value, quality factor and status.

To modify behaviour for this command, modify the do() method of the command class.

Parameters
argin – Name of the alarm

Returns
JSON string containing alarm data

GetAlarmAdditionalInfo(*argin*)

Get additional alarm information.

To modify behaviour for this command, modify the do() method of the command class.

Parameters
argin – Name of the alarm

Returns
JSON string containing additional alarm information

GetAlarmStats()

Get current alarm stats.

To modify behaviour for this command, modify the do() method of the command class.

Returns
JSON string containing alarm statistics

GetAlertStats()

Get current alert stats.

To modify behaviour for this command, modify the do() method of the command class.

Returns
JSON string containing alert statistics

ska_tango_base.alarm_handler_device.**main**(*args=None*, ***kwargs*)

Launch an SKAAlarmHandler device.

2.6 Capability Device

SKACapability.

Capability handling device

class ska_tango_base.capability_device.**SKACapability**(**args: Any*, ***kwargs: Any*)

A Subarray handling device.

It exposes the instances of configured capabilities.

init_command_objects()

Set up the command objects.

class InitCommand(*target*, *op_state_model*, *logger=None*)

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

do()

Stateless hook for device initialisation.

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

Return type
(*ResultCode*, str)

activationTime = tango.server.attribute(dtype=double, unit=s, standard_unit=s, display_unit=s, doc=Time of activation in seconds since Unix epoch.)

Device attribute.

configuredInstances = tango.server.attribute(dtype=uint16, doc=Number of instances of this Capability Type currently in use on this subarray.)

Device attribute.

usedComponents = tango.server.attribute(dtype=('str',), max_dim_x=100, doc=A list of components with no. of instances in use on this Capability.)

Device attribute.

always_executed_hook()

Perform actions that are executed before every device command.

This is a Tango hook.

delete_device()

Clean up any resources prior to device deletion.

This method is a Tango hook that is called by the device destructor and by the device Init command. It allows for any memory or other resources allocated in the init_device method to be released prior to device deletion.

read_activationTime()

Read time of activation since Unix epoch.

Returns
Activation time in seconds

read_configuredInstances()

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

Returns
The number of configured instances of a capability in a subarray

read_usedComponents()

Read the list of components with no.

of instances in use on this Capability :return: The number of components currently in use.

class ConfigureInstancesCommand(*target*, **args*, *logger=None*, ***kwargs*)

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

do(*argin*)

Stateless hook for ConfigureInstances()) command functionality.

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

Return type
(*ResultCode*, str)

ConfigureInstances(*argin*)

Specify the number of instances of the current capacity to be configured.

To modify behaviour for this command, modify the do() method of the command class.

Parameters
argin – Number of instances to configure

Returns
None.

ska_tango_base.capability_device.**main**(*args=None*, ***kwargs*)

Launch an SKACapability device.

2.7 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 ska_tango_base.logger_device.**SKALogger**(**args: Any*, ***kwargs: Any*)

A generic base device for Logging for SKA.

init_command_objects()

Set up the command objects.

always_executed_hook()

Perform actions that are executed before every device command.

This is a Tango hook.

delete_device()

Clean up any resources prior to device deletion.

This method is a Tango hook that is called by the device destructor and by the device Init command. It allows for any memory or other resources allocated in the init_device method to be released prior to device deletion.

class SetLoggingLevelCommand(*target*, *state_model*, *logger=None*)

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

do(*argin*)

Stateless hook for SetLoggingLevel() command functionality.

Returns

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

Return type

(*ResultCode*, str)

SetLoggingLevel(*argin*)

Set the logging level of the specified devices.

To modify behaviour for this command, modify the do() method of the command class.

Parameters

argin (tango.DevVarLongStringArray) – Array consisting of

- argin[0]: list of DevLong. Desired logging level.
- argin[1]: list of DevString. Desired tango device.

Returns

None.

ska_tango_base.logger_device.**main**(*args=None*, ***kwargs*)

Launch an SKALogger device.

2.8 Controller Device

SKAController.

Controller device

class ska_tango_base.controller_device.**SKAController**(**args: Any*, ***kwargs: Any*)

Controller device.

init_command_objects()

Set up the command objects.

class **InitCommand**(*target*, *op_state_model*, *logger=None*)

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

do()

Stateless hook for device initialisation.

Returns

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

Return type

(*ResultCode*, str)

elementLoggerAddress = tango.server.attribute(dtype=str, doc=FQDN of Element Logger)

Device attribute.

elementAlarmAddress = tango.server.attribute(dtype=str, doc=FQDN of Element Alarm Handlers)

Device attribute.

elementTelStateAddress = tango.server.attribute(dtype=str, doc=FQDN of Element TelState device)

Device attribute.

elementDatabaseAddress = tango.server.attribute(dtype=str, doc=FQDN of Element Database device)

Device attribute.

maxCapabilities = tango.server.attribute(dtype=('str',), max_dim_x=20, doc=Maximum number of instances of each capability type, e.g. 'CORRELATOR:512', 'PSS-BEAMS:4'.)

Device attribute.

availableCapabilities = tango.server.attribute(dtype=('str',), max_dim_x=20, doc=A list of available number of instances of each capability type, e.g. 'CORRELATOR:512', 'PSS-BEAMS:4'.)

Device attribute.

always_executed_hook()

Perform actions that are executed before every device command.

This is a Tango hook.

delete_device()

Clean up any resources prior to device deletion.

This method is a Tango hook that is called by the device destructor and by the device Init command. It allows for any memory or other resources allocated in the init_device method to be released prior to device deletion.

read_elementLoggerAddress()

Read FQDN of Element Logger device.

read_elementAlarmAddress()

Read FQDN of Element Alarm device.

read_elementTelStateAddress()

Read FQDN of Element TelState device.

read_elementDatabaseAddress()

Read FQDN of Element Database device.

read_maxCapabilities()

Read maximum number of instances of each capability type.

read_availableCapabilities()

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

class IsCapabilityAchievableCommand(*target*, **args*, *logger=None*, ***kwargs*)

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

do(*argin*)

Stateless hook for device IsCapabilityAchievable() command.

Returns

Whether the capability is achievable

Return type

bool

isCapabilityAchievable(*argin*)

Check if provided capabilities can be achieved by the resource(s).

To modify behaviour for this command, modify the do() method of the command class.

Parameters

argin (tango.DevVarLongStringArray.) – An array consisting pair of

- [nrInstances]: DevLong. Number of instances of the capability.
- [Capability types]: DevString. Type of capability.

Returns

True if capability can be achieved, False if cannot

Return type

DevBoolean

ska_tango_base.controller_device.**main**(*args=None*, ***kwargs*)

Launch an SKAController Tango device.

2.9 Tel State Device

SKATelState.

A generic base device for Telescope State for SKA.

class ska_tango_base.tel_state_device.**SKATelState**(**args: Any*, ***kwargs: Any*)

A generic base device for Telescope State for SKA.

always_executed_hook()

Perform actions that are executed before every device command.

This is a Tango hook.

delete_device()

Clean up any resources prior to device deletion.

This method is a Tango hook that is called by the device destructor and by the device Init command. It allows for any memory or other resources allocated in the init_device method to be released prior to device deletion.

ska_tango_base.tel_state_device.**main**(*args=None*, ***kwargs*)

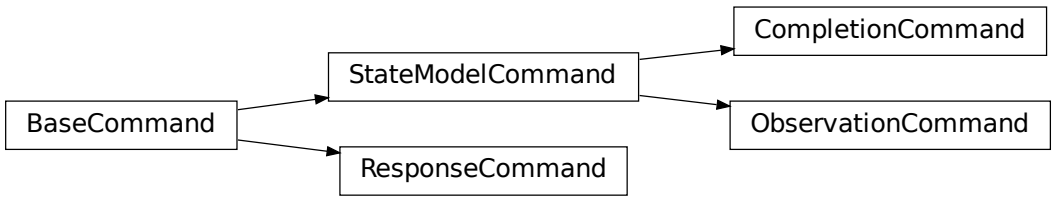
Launch an SKATelState device.

2.10 Commands

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

The following command classes are provided:

- **BaseCommand**: that implements the common pattern for commands; implement the do() method, and invoke the command class by *calling* it.
- **StateModelCommand**: implements a command that drives a state model. For example, a command that drives the operational state of the device, such as `On()`, `Standby()` and `Off()`, is a `StateModelCommand`.
- **ObservationCommand**: implements a command that drives the observation state of an obsDevice, such as a subarray; for example, `AssignResources()`, `Configure()`, `Scan()`.
- **ResponseCommand**: for commands that return a `(ResultCode, message)` tuple.
- **CompletionCommand**: for commands that need to let their state machine know when they have completed; that is, long-running commands with transitional states, such as `AssignResources()` and `Configure()`.



Multiple inheritance is supported, and it is expected that many commands will need to inherit from more than one command class. For example, a subarray's `AssignResources` command would inherit from:

- `ObservationState`, because it drives observation state
- `ResponseCommand`, because it returns a *(ResultCode, message)*
- `CompletionCommand`, because it needs to let its state machine know when it is completed.

To use these commands: subclass from the command classes needed, then implement the `__init__` and `do` methods. For example:

```
class AssignResourcesCommand(
    ObservationCommand, ResponseCommand, CompletionCommand
):
    def __init__(self, target, op_state_model, obs_state_model, logger=None):
        super().__init__(target, obs_state_model, "assign", op_state_model,␣
↪logger=logger)

    def do(self, argin):
        # do stuff
        return (ResultCode.OK, "AssignResources command completed OK")
```

class ska_tango_base.commands.**ResultCode**(*value*)

Python enumerated type for command return 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.

class ska_tango_base.commands.**BaseCommand**(*target*, **args*, *logger=None*, ***kwargs*)

Abstract base class for Tango device server commands.

Checks that the command is allowed to run in the current state, and runs the command.

do(*argin=None*)

Perform the user-specified functionality of the command.

This class provides stub functionality; subclasses should subclass this method with their command functionality.

Parameters

argin (*ANY*) – the argument passed to the Tango command, if present

class ska_tango_base.commands.**StateModelCommand**(*target*, *state_model*, *action_slug*, **args*, *logger=None*, ***kwargs*)

A base class for commands that drive a state model.

is_allowed(*raise_if_disallowed=False*)

Whether this command is allowed to run in the current state of the state model.

Parameters
raise_if_disallowed – whether to raise an error or simply return False if the command is disallowed

Returns
whether this command is allowed to run

Return type
boolean

Raises
CommandError – if the command is not allowed and *raise_if_disallowed* is True

class ska_tango_base.commands.**ObservationCommand**(*target*, *obs_state_model*, *action_slug*, *op_state_model*, **args*, *logger=None*, ***kwargs*)

A base class for commands that drive the device's observing state.

This is a special case of a `StateModelCommand` because although it only drives the observation state model, it has to check also the operational state model to determine whether it is allowed to run.

is_allowed(*raise_if_disallowed=False*)

Whether this command is allowed to run in the current state of the state model.

Parameters
raise_if_disallowed – whether to raise an error or simply return False if the command is disallowed

Returns
whether this command is allowed to run

Return type
boolean

Raises
CommandError – if the command is not allowed and *raise_if_disallowed* is True

class ska_tango_base.commands.**ResponseCommand**(*target*, **args*, *logger=None*, ***kwargs*)

A command returns a (ResultCode, message) tuple.

This is an Abstract base class for commands that execute a procedure or operation, then return a (ResultCode, message) tuple.

class ska_tango_base.commands.**CompletionCommand**(*target*, *state_model*, *action_slug*, **args*, *logger=None*, ***kwargs*)

A command that triggers an action on the state model at completion.

This is an abstract base class for commands that need to signal completion by triggering a "completed" action on the state model.

completed()

Let the state model know that the command has completed.

2.11 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.

class ska_tango_base.control_model.**HealthState**(*value*)

Python enumerated type for `healthState` attribute.

OK = 0

Tango Device reports this state when ready for use, or when entity `adminMode` is `NOT_FITTED` or `RESERVED`.

The rationale for reporting health as OK when an entity is `NOT_FITTED` or `RESERVED` is to ensure that it does not pop-up unnecessarily on drill-down fault displays with healthState `UNKNOWN`, `DEGRADED` or `FAILED` while it is expected to not be available.

DEGRADED = 1

Tango Device reports this state when only part of functionality is available. This value is optional and shall be implemented only where it is useful.

For example, a subarray may report healthState as `DEGRADED` if one of the dishes that belongs to a subarray is unresponsive, or may report healthState as `FAILED`.

Difference between `DEGRADED` and `FAILED` health shall be clearly identified (quantified) and documented. For example, the difference between `DEGRADED` and `FAILED` subarray can be defined as the number or percent of the dishes available, the number or percent of the baselines available, sensitivity, or some other criterion. More than one criteria may be defined for a Tango Device.

FAILED = 2

Tango Device reports this state when unable to perform core functionality and produce valid output.

UNKNOWN = 3

Initial state when health state of entity could not yet be determined.

class ska_tango_base.control_model.**AdminMode**(*value*)

Python enumerated type for `adminMode` attribute.

ONLINE = 0

The component can be used for normal operations, such as observing.

While in this mode, the Tango device is actively monitoring and controlling its component. Tango devices that implement `adminMode` as a read-only attribute shall always report `adminMode=ONLINE`.

OFFLINE = 1

The component is not to be used for any operations.

While in this mode, Tango devices report `state=DISABLE`, and do not communicate with their component. Monitoring and control of the component does not occur, so alarms, alerts and events are not received.

MAINTENANCE = 2

SKA operations declares that the component cannot be used for normal operations, but can be used for maintenance purposes such as testing and debugging, as part of a "maintenance subarray". While in this mode, Tango devices are actively monitoring and controlling their component, but may only support a subset of normal functionality.

MAINTENANCE mode has different meaning for different components, depending on the context and functionality. Some entities may implement different behaviour when in MAINTENANCE mode. For each Tango device, the difference in behaviour and functionality in MAINTENANCE mode shall be documented.

NOT_FITTED = 3

The component cannot be used for any purposes because it is not fitted; for example, faulty equipment has been removed and not yet replaced, leaving nothing *in situ* to monitor.

While in this mode, Tango devices report state=DISABLED. All monitoring and control functionality is disabled because there is no component to monitor.

RESERVED = 4

The component is fitted, but only for redundancy purposes.

It is additional equipment that does not take part in operations at this time, but is ready to take over when the operational equipment fails. While in this mode, Tango devices report state=DISABLED. All monitoring and control functionality is disabled.

class ska_tango_base.control_model.**ObsState**(*value*)

Python enumerated type for obsState attribute - the observing state.

EMPTY = 0

The sub-array is ready to observe, but is in an undefined configuration and has no resources allocated.

RESOURCING = 1

The system is allocating resources to, or deallocating resources from, the subarray.

This may be a complete de/allocation, or it may be incremental. In both cases it is a transient state and will automatically transition to IDLE when complete. For some subsystems this may be a very brief state if resourcing is a quick activity.

IDLE = 2

The subarray has resources allocated and is ready to be used for observing.

In normal science operations these will be the resources required for the upcoming SBI execution.

CONFIGURING = 3

The subarray is being configured ready to scan.

On entry to the state no assumptions can be made about the previous conditions. It is a transient state and will automatically transition to READY when it completes normally.

READY = 4

The subarray is fully prepared to scan, but is not actually taking data or moving in the observed coordinate system (it may be tracking, but not moving relative to the coordinate system).

SCANNING = 5

The subarray is taking data and, if needed, all components are synchronously moving in the observed coordinate system.

Any changes to the sub-systems are happening automatically (this allows for a scan to cover the case where the phase centre is moved in a pre-defined pattern).

ABORTING = 6

The subarray is trying to abort what it was doing due to having been interrupted by the controller.

ABORTED = 7

The subarray has had its previous state interrupted by the controller, and is now in an aborted state.

RESETTING = 8

The subarray device is resetting to the IDLE state.

FAULT = 9

The subarray has detected an error in its observing state making it impossible to remain in the previous state.

RESTARTING = 10

The subarray device is restarting, as the last known stable state is where no resources were allocated and the configuration undefined.

class ska_tango_base.control_model.**ObsMode**(*value*)

Python enumerated type for `obsMode` attribute - the observing mode.

IDLE = 0

The `obsMode` shall be reported as `IDLE` when `obsState` is `IDLE`; else, it will correctly report the appropriate value.

More than one observing mode can be active in the same subarray at the same time.

IMAGING = 1

Imaging observation is active.

PULSAR_SEARCH = 2

Pulsar search observation is active.

PULSAR_TIMING = 3

Pulsar timing observation is active.

DYNAMIC_SPECTRUM = 4

Dynamic spectrum observation is active.

TRANSIENT_SEARCH = 5

Transient search observation is active.

VLBI = 6

Very long baseline interferometry observation is active.

CALIBRATION = 7

Calibration observation is active.

class ska_tango_base.control_model.**ControlMode**(*value*)

Python enumerated type for `controlMode` attribute.

REMOTE = 0

Tango Device accepts commands from all clients.

LOCAL = 1

Tango Device accepts only from a 'local' client and ignores commands and queries received from TM or any other 'remote' clients. This is typically activated by a switch, or a connection on the local control interface. The intention is to support early integration of DISHes and stations. The equipment has to be put back in `REMOTE` before clients can take control again. `controlMode` may be removed from the SCM if unused/not needed.

Note: Setting *controlMode* to *LOCAL* **is not a safety feature**, but rather a usability feature. Safety has to be implemented separately to the control paths.

class ska_tango_base.control_model.**SimulationMode**(*value*)

Python enumerated type for simulationMode attribute.

FALSE = 0

A real entity is connected to the control system.

TRUE = 1

A simulator is connected to the control system, or the real entity acts as a simulator.

class ska_tango_base.control_model.**TestMode**(*value*)

Python enumerated type for testMode attribute.

This enumeration may be replaced and extended in derived classes to add additional custom test modes. That would require overriding the base class testMode attribute definition.

NONE = 0

Normal mode of operation.

No test mode active.

TEST = 1

Element (entity) behaviour and/or set of commands differ for the normal operating mode.

To be implemented only by devices that implement one or more test modes. The Element documentation shall provide detailed description.

class ska_tango_base.control_model.**LoggingLevel**(*value*)

Python enumerated type for loggingLevel attribute.

class ska_tango_base.control_model.**PowerMode**(*value*)

Enumerated type for power mode.

Used by components that rely upon a power supply, such as hardware.

2.12 Faults

General SKA Tango Device Exceptions.

exception ska_tango_base.faults.**SKABaseError**

Base class for all SKA Tango Device exceptions.

exception ska_tango_base.faults.**GroupDefinitionsError**

Error parsing or creating groups from GroupDefinitions.

exception ska_tango_base.faults.**LoggingLevelError**

Error evaluating logging level.

exception ska_tango_base.faults.**LoggingTargetError**

Error parsing logging target string.

exception ska_tango_base.faults.**ResultCodeError**

A method has returned an invalid return code.

exception ska_tango_base.faults.**StateModelError**

Error in state machine model related to transitions or state.

exception ska_tango_base.faults.**CommandError**
Error executing a BaseCommand or similar.

exception ska_tango_base.faults.**CapabilityValidationError**
Error in validating capability input against capability types.

exception ska_tango_base.faults.**ComponentError**
Component cannot perform as requested.

exception ska_tango_base.faults.**ComponentFault**
Component is in FAULT state and cannot perform as requested.

2.13 Release

Release information for ska_tango_base Python Package.

2.14 Utils

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

ska_tango_base.utils.**exception_manager**(*cls*, *callback=None*)
Return a context manager that manages exceptions.

ska_tango_base.utils.**get_dev_info**(*domain_name*, *device_server_name*, *device_ref*)
Get device info.

ska_tango_base.utils.**dp_set_property**(*device_name*, *property_name*, *property_value*)
Use a DeviceProxy to set a device property.

ska_tango_base.utils.**get_device_group_and_id**(*device_name*)
Return the group and id part of a device name.

ska_tango_base.utils.**convert_api_value**(*param_dict*)
Validate tango command parameters which are passed via json.

Parameters
param_dict –

Returns

ska_tango_base.utils.**coerce_value**(*value*)
Coerce tango.DevState values to string, leaving other values alone.

ska_tango_base.utils.**get_dp_attribute**(*device_proxy*, *attribute*, *with_value=False*, *with_context=False*)
Get an attribute from a DeviceProxy.

ska_tango_base.utils.**get_dp_command**(*device_name*, *command*, *with_context=False*)
Get a command from a DeviceProxy.

ska_tango_base.utils.**get_tango_device_type_id**(*tango_address*)
Return the type id of a TANGO device.

ska_tango_base.utils.**get_groups_from_json**(*json_definitions*)

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",
```

(continues on next page)

(continued from previous page)

```
                "devices": [
                    "elt/server/3",
                    "elt/server/4",
                    "elt/switch/B",
                    "elt/pdu/rackB"
                ],
                "subgroups": []
            }
        ]
    }
]
```

Parameters

json_definitions (*sequence of str*) – 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.

Return type

dict

Raises

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]

ska_tango_base.utils.**validate_capability_types**(*command_name*, *requested_capabilities*, *valid_capabilities*)

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

Parameters

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

ska_tango_base.utils.**validate_input_sizes**(*command_name*, *argin*)

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

Parameters

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

ska_tango_base.utils.**convert_dict_to_list**(*dictionary*)

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

ska_tango_base.utils.**for_testing_only**(*func*, *_testing_check=<function <lambda>>*)

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):
    ...
```

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

INDEX

A

D

E

F

G

L

M

N

O

S

T

U

V

W