# CSP.LMC Common Software Documentation

***Release 1.0.1***

**SKA Organization**

**Jan 25, 2023**

# CSP.LMC COMMON PACKAGE:

CHAPTER

ONE

# CSP.LMC COMMON PACKAGE DESCRIPTION

General requirements for the monitor and control functionality are the same in both telescopes. In addition two of three other CSP Sub-elements, namely PSS and PST, have the same functionality and use the same design for both the telescopes.

Functionality common to Low and Mid CSP.LMC includes: communication framework, logging, archiving, alarm generation, subarraying, some of the functionality realated to handling observing mode changes, Pulsar Search and Pulsar Timing, and to some extent Very Long Baseline Interferometry (VLBI).

The difference between LOW and MID CSP.LMC is mostly due to the different receivers (dishes vs stations) and different CBF functionality and design. More than the 50% of the CSP.LMC functionality is common for both telescopes.

The CSP.LMC Common Package comprises all the software components and functionality common to LOW and MID CSP.LMC and is used as a base for development of the Low CSP.LMC and Mid CSP.LMC software.

The *CSP.LMC Common Package* is delivered as a part of each CSP.LMC release, via a Python package that can be used as required for maintenance and upgrade.

CSP.LMC implements a high level interface (API) that Telescope Manager (TM), or other authorized client, can use to monitor and control CSP as a single instrument.

At the same time, CSP.LMC provides high level commands that the TM can use to sub-divide the array into up to 16 sub-arrays, i.e. to assign station/receptors to sub-arrays, and to operate each sub-array independently and concurrently with all other sub-arrays.

The top-level software components provided by CSP.LMC API are:

- *Csp Controller*
- *Csp Subarray*
- CSP Alarm Handler (TBD)
- CSP Logger (TBD)
- CSP TANGO Facility Database (TBD)
- Input processor Capability (receptors/stations) (TBD)
- *Search Beam Capability* (TBD)
- *Timing Beam Capability* (TBD)
- *VLBI Beam Capability* (TBD)

Components listed above are implemented as TANGO devices, i.e. classes that implement standard TANGO API. The CSP.LMC TANGO devices are based on the standard SKA1 TANGO Element Devices provided via the *SKA Base Classes package*.

## 1.1 CSP.LMC Controller

The CSP controller provides API for monitor and control the CSP sub-system. CSP Controller is the primary point of access for CSP Monitor and Control.

CSP Controller maintains the pool of schedulable resources, and it can relies on the CSP CapabilityMonitor devices, as needed. The CSP Controller implements CSP sub-system-level status indicators, configuration parameters, housekeeping commands.

## 1.2 CSP.LMC Subarray

The core CSP functionality, configuration and execution of signal processing, is configured, controlled and monitored via subarrays.

CSP Subarray makes provision to TM to configure a subarray, select Processing Mode and related parameters, specify when to start/stop signal processing and/or generation of output products. TM accesses directly a CSP Subarray to:

- Assign resources
- Configure a scan
- Control and monitor states/operations

### 1.2.1 Resources assignment

The assignment of Capabilities to a subarray (*subarray composition*) is performed in advance of a scan configuration. Assignable Capabilities for CSP Subarrays are:

- receptors (MID) or stations (LOW)
- tied-array beams: Search Beams, Timing Beams and Vlbi Beams.

In general resource assignment to a subarray is exclusive, but in some cases the same Capability instance may be used in shared manner by more then one subarray.

### 1.2.2 Inherent Capabilities

Each CSP subarray has also a set of permanently assigned *inherent Capabilities*: the number and type is different for LOW and MID instance.

Only the Inherent Capabilities related to the Processing Mode are common to both instances.

These are:

- Correlation
- PSS
- PST
- VLBI

An inherent Capability can be enabled or disabled, but cannot assigned or removed to/from a subarray.

### 1.2.3 Scan configuration

TM provides a complete scan configuration to a subarray via an ASCII JSON encoded string. Parameters specified via a JSON string are implemented as TANGO Device attributes and can be accessed and modified directly using the buil-in TANGO method *write_attribute*. When a complete and coherent scan configuration is received and the subarray configuration (or re-configuration) completed, the subarray it's ready to observe.

### 1.2.4 Control and Monitoring

Each CSP Subarray maintains and report the status and state transitions for the CSP subarray as a whole and for individual assigned resources.

In addition to pre-configured status reporting, a CSP subarray makes provision for the TM and any authorized client, to obtain the value of any subarray attribute.

## 1.3 CSP.LMC Capabilities

Capabilities represent the CSP schedulable resources and provide API that can be used to configure, monitor and control resources that implement signal processing functionality. During normal operations, TM uses the sub-array API to assign capabilities to the sub-array, configure sub-array Processing Mode, start and stop scan.

The CSP.LMC Common Package implements the capabilities that are shared between LOW and MID instances.

These are:

- *CSP Search Beam Capability*
- *CSP Timing Beam Capability*
- *CSP VLBI Beam Capability*

### 1.3.1 CSP.LMC Search Beam Capability

(To be implemented)

The Search Beam Capability exposes the attributes and commands to monitor and control beam-forming and PSS processing in a single beam.

The mapping between an instance of the CSP Search Beam and the internal CSP Sub-element components performing beam-forming and search is established at initialization and is permanent.

#### CSP.LMC SearchBeamCapability API Documentation

(To be implemented)

### 1.3.2 CSP.LMC Timing Beam Capability

(To be implemented)

The Timing Beam Capability exposes the attributes and commands to monitor and control beam-forming and PST processing in a single beam.

The mapping between an instance of the CSP Search Beam and the internal CSP Sub-element components performing beam-forming and search is established at initialization and is permanent.

#### CSP.LMC TimingBeamCapability API Documentation

(To be implemented)

### 1.3.3 CSP.LMC VLBI Beam Capability

(To be implemented)

The VLBI Beam Capability exposes the attributes and commands to monitor and control beamforming and VLBI processing in a single beam.

#### CSP.LMC VlbiBeamCapability API Documentation

### 1.3.4 CSP.LMC CapabilityMonitor

(To be implemented)

#### CSP.LMC CapabilityMonitor API Documentation

(To be implemented)

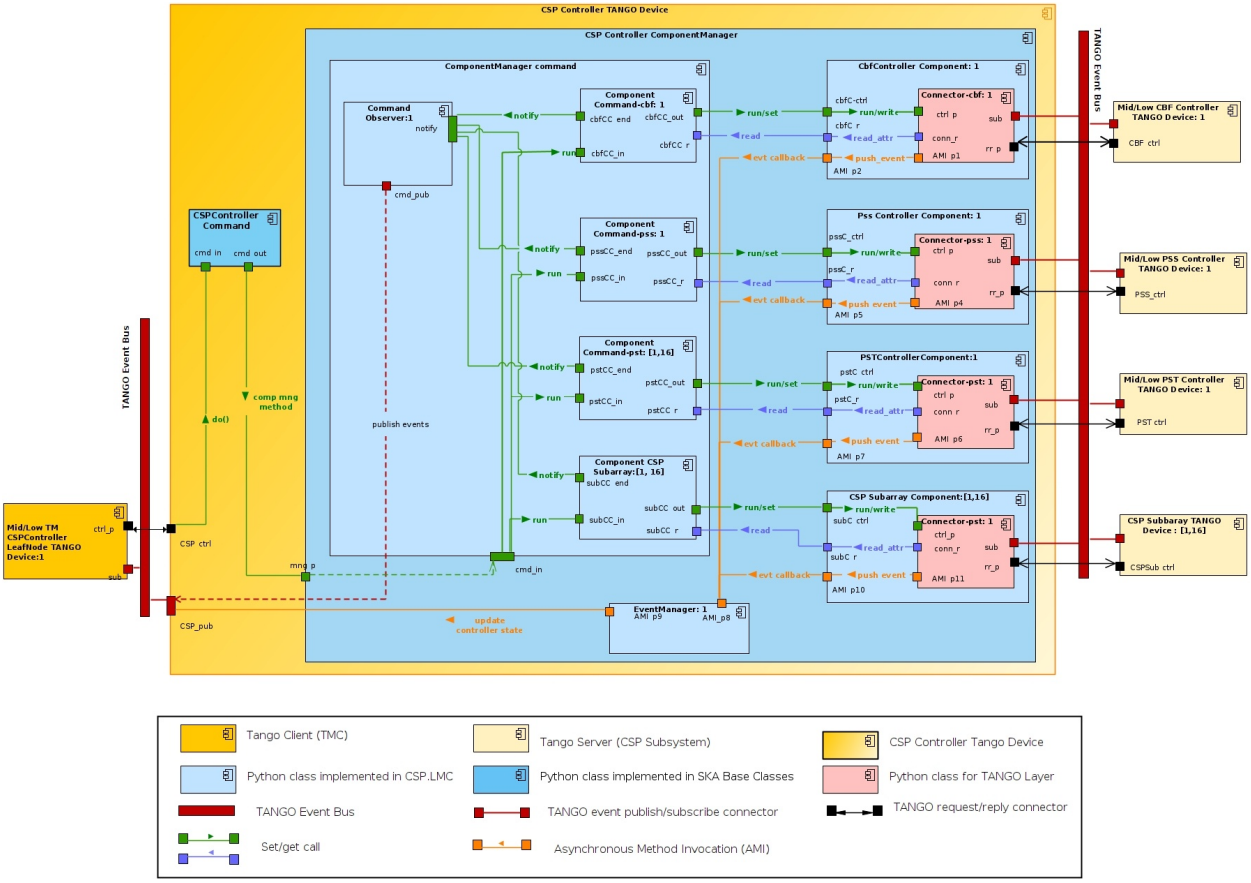
CHAPTER

TWO

# ARCHITECTURE DESCRIPTION

The architecture of CSP.LMC is shared between the Controller and the Subarray. Both of them communicate with three sub-systems: CBF, PSS and PST. The Controller must also access the CSP.LMC Subarrays and the Capabilities device manager to report the information on the resources.

In the figure below, the C&C view of CSP.LMC controller is provided. The case of CSP.LMC subarray is identical, where Subsystem's Controller are substituted with correspondant Subsystem's subarrays and no other CSP.LMC subarrays are controlled. Further diagrams and a more comprehensive description of its component can be found at this page.




The main operations of CSP are carried out into the three sub-elements. The interaction between the CSP Controller and the subordinate sub-systems devices are mediated through a Python class that works as a proxy (Component Class). This approach has the advantage of abstraction.

Since version 0.11.0 the state machine of ska-tango-base is no longer used. The motivation of this choice is described

here. For this reason, custom state models are implemented for the Operational, Observing and Health State (*CSP State Models*).

## 2.1 Interface to subsystem TANGO devices

Specific operations on a sub-element can be done by specializing the proxy class for each sub-system and the corresponding functions are maintained in a specific part of the code.

### 2.1.1 Csp sub-system Component

The component class is a mediator between CSP.LMC and a Subsystem Device. It acts as an adapter and allows, when needed, to execute specific instructions on a subsystem before invoking the required command. In other words, its functionalities are:

- read and write of associated device's attributes;
- command execution;
- subscription of attributes on the corresponding Tango Device.

### 2.1.2 Connector

Connector Class is class working as interface to the TANGO system. It relies on TangoClient class of ska-tmc-common package developed by NCRA team, and it has the purpose to communicate with the device proxy of Sub System TANGO device for all the functionalities used by the Component classe.

One of the main advantage to have this class, it the possibility to be easily mocked during the tests.

## 2.2 Commands execution

A command issued on the CSP Controller or CSP Subarray (controller command) by a TANGO client or the TM, breaks up, nearly always, into several commands (>=3), one for each CSP sub-system. These commands (sub-commands or component commands) are forwarded to the connected sub-sub-system.

The CSP Controller or Subarray TANGO device has to be able to invoke the command on a sub-element and monitors its execution, detecting its progress and its final status (success/failure).

The sequence of operation to be performed are the following:

- check the initial device state to determine if the command is allowed;
- wait for the final status (the one expected after the end of successful execution) and detect possible conditions of failures;
- implement support for timeout;
- report the end of the command.

The execution of a command is reported by the attribute CommandResult, which is a Tuple with the name of the latest command invoked and a the resultCode ENUM (from ska-tango-base) that report the state of the command (SUCCEEDED:0, STARTED:1, FAILED:3)

### 2.2.1 Command Observer

A specific Python class (CommandObserver Class), using the Observer Pattern Design, is used to detect the controller command completion. Each component command is registered within the observer and notifies it when it has completed

A CSP Subarray command is considered completed when all the forwarded commands have ended. This component monitors the execution of a CSP Subarray command, keeping track of the commands running on the CSP sub-systems.

When the execution of a command ends on a sub-system, the Component sub-system notifies this condition to the CommandObserver invoking the notify method provided by this component.

This component implements also a set of attributes to report information about the status of each monitored sub-system command, as for example the running and progress status.

At the end of the command, the Command Observer report the status of the command to the commandResult attribute.

### 2.2.2 Sub-system Command (Component Command)

The Component Command models a command acting on a sub-system Component instance. It implements the logic to manage and control the command issued on a single component. The ComponentCommand class, when instantiated for a specific command (On, Off , etc) contains all the information about the request such as:

- the input parameters (if any)
- the Component to act on
- success, failure and timeout conditions

When the CSP Controller invokes the run method, each Component Command will run one (or more actions) on the associated Component object. When the Command ends, it reports to the Command Observer the success or the failure.

## 2.3 Event Manager

Management of the events is delegated to a specific class (Event Manager Class). On initialization completion (when the connection with the sub-system devices has been established) CSP.LMC devices (Controller and Subarray) select which events are to be monitored on the sub-systems and delegate the subscription to the EventManager. The aim of this class is to aggregate and report to TM the collective states and modes of the CSP (State, ObsState, HealthState, ecc…).

In other words, Event Manager works on the behalf of the CSP.LMC to:

- subscribes the events for the main state and modes subsystem's attributes (registering callbacks to the Component's classes);
- retrieve the value or errors reported by the callback registered with the events
- carry out particular policies of aggregation on attributes, reducing the load of information traveling to the subarray;

This object does not subscribe directly to a sub-system TANGO devices, but relies on the corresponding Component objects to perform such work. The events received from each sub-system are pushed back to the CSP Subarray via callbacks registered at subscription time.

CHAPTER

# THREE

# PROJECT'S API

## 3.1 CSP.LMC Common Devices API

### 3.1.1 CspController

**class** ska_csp_lmc_common.controller_device.**CspController**(**args: Any*, ***kwargs: Any*)

Bases: SKAController

CSP Controller functionality is modeled via a Common Class for the CSP Controller TANGO Devices (Mid and Low).

**Device Properties:**

**ConnectionTimeout**

- The maximum time to wait for the connection with a sub-system
- Type:'DevUShort'
- Default Value: 60

**PingConnectionTime**

- The time to wait between connection attempts
- Type:'DevUShort'
- Default Value: 5

**CspCbf**

- FQDN of the CBF Sub-element Master
- Type:'DevString'

**CspPss**

- TANGO FQDN of the PSS Sub-element Master
- Type:'DevString'

**CspPstBeams**

- TANGO FQDNs of the PST Beams
- Type:'DevVarStringArray'

**CspSubarrays**

- TANGO FQDN of the CSP.LMC Subarrays

- Type:'DevVarStringArray'

**SearchBeamsMonitor**

- TANGO Device to monitor the CSP SearchBeams Capability devices.
- Type:'DevString'

**TimingBeamsMonitor**

- TANGO Device to monitor the CSP TimingBeams Capability devices.
- Type:'DevString'

**VlbiBeamsMonitor**

- TANGO Device to monitor the CSP VlbiBeams Capability devices.
- Type:'DevString'

`init_device()` → None

Override the Base Classes *init_device* method to change the asynchronous callback sub-model from pull to push sub-model.

`_init_state_model()` → None

Override the health and operational State models:

current CSP.LMC implementation does no longer rely on the SKA State models and associated State Machine library.

`delete_device()` → None

Hook to delete resources allocated in init_device.

This method allows for any memory or other resources allocated in the init_device method to be released. This method is called by the device destructor and by the device Init command.

`set_component_manager`(*cm_configuration: ComponentManagerConfiguration*) → CSPControllerComponentManager

Configure the ComponentManager for the CSP Controller device. This method has to be specialized in Mid.CSP and Low.CSP.

**Parameters**
**cm_configuration** – A class with all the device properties accessible as attributes

**Returns**
The CSP Controller ComponentManager

`create_component_manager()` → CSPControllerComponentManager

Override the base method.

**Returns**
The CSP ControllerComponentManager

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

Bases: `InitCommand`

Class for the controller Init command.

`do()`

Invoke the *init* command on the Controller ComponentManager.

**Returns**
a tuple with the command result code and an informative message.

**update_ctrl_state**(*value: tango.DevState*) → None

Update the CSP Controller state to the value returned by the Operational State Model.

**Parameters**
**value** – the CSP Controller state calculated as an aggregation of the CSP Controller subsystems states.

**update_ctrl_health_state**(*value: ska_tango_base.control_model.HealthState*) → None

Update the CSP Controller healthState as aggregation of the CSP sub- systems' health states.

**Parameters**
**value** – the updated health state value

**update_property**(*property_name: str*, *property_value: Any*) → None

General method invoked by the ComponentManager to push an event on a device attribute properly configured to push events from the device.

**Parameters**

- **property_name** – the TANGO attribute name
- **property_value** – the attribute value

**set_state**(*state*)

Put the state change into the internal queue.

**set_status**(*status*)

Put the state change into the internal queue.

**push_change_event**(*name*, *value=None*)

Put the change event into the internal queue.

**PushChanges**()

Periodic command to push events from the device.

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

Bases: ResponseCommand

Class to implement the main functionalities of the CSP Controller On command.

**is_allowed**(*raise_if_disallowed: bool = True*) → bool

Method invoked by the TANGO method is_On_allowed() to determine whether the On command can be invoked on the device.

**Parameters**
**raise_if_disallowed** – whether to raise an error or simply return False if the command is disallowed. Currently not used.

**Returns**
whether this command is allowed to run

**do**(*argin: List[str]*) → Tuple[ska_tango_base.commands.ResultCode, str]

Invoke the *on* command on the Controller ComponentManager.

**Parameters**
**argin** – A list of TANGO Device fqdn to switch on.

**Returns**
a tuple with the command result code and an informative message.

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

Bases: ResponseCommand

Class for the controller Off command.

The class inherits from the base ResponseCommand because current implementation does not rely on the SKA Operational State Model.

**is_allowed**(*raise_if_disallowed: bool = True*) → bool

Method invoked by the TANGO method is_Off_allowed() to determine whether the On command can be invoked on the device.

**Parameters**
**raise_if_disallowed** – whether to raise an error or simply return False if the command is disallowed. Currently not used.

**Returns**
whether this command is allowed to run

**do**(*argin: List[str]*) → Tuple[ska_tango_base.commands.ResultCode, str]

Invoke the *off* command on the Controller ComponentManager.

**Parameters**
**argin** – A list of TANGO Device fqdn to switch off.

**Returns**
a tuple with the command result code and an informative message.

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

Bases: ResponseCommand

Class for the controller Standby command.

The class inherits from the base ResponseCommand because current implementation does not rely on the SKA Operational State Model.

**is_allowed**(*raise_if_disallowed: bool = True*) → bool

Method invoked by the TANGO method *is_Standby_allowed()* to determine whether the Standby command can be invoked on the device.

The Standby command is allowed from the following operational state: ON, OFF, DISABLE, FAULT

**Parameters**
**raise_if_disallowed** – whether to raise an error or simply return False if the command is disallowed. *Currently not used.*

**Returns**
whether this command is allowed to run

**do**(*argin: List[str]*) → Tuple[ska_tango_base.commands.ResultCode, str]

Invoke the *standby* command on the Controller ComponentManager.

**Parameters**
**argin** – A list of TANGO Device fqdn to set in low-power mode.

**Returns**
a tuple with the command result code and an informative message.

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

Bases: ResponseCommand

Class for the Reset command.

The class inherits from the base ResponseCommand because current implementation does not rely on the SKA Operational State Model.

**is_allowed**(*raise_if_disallowed: bool = True*) → bool

Method invoked by the TANGO method *is_Reset_allowed()* to determine whether the Reset command can be invoked on the device.

The *Reset* command is allowed from any state except INIT: in this way, even if the CSP Controller is not in FAULT, it is possible to reset the faulty condition of a CSP sub-system.

**Parameters**
- **raise_if_disallowed** – whether to raise an error or simply return False if the command is disallowed.
- **raise_if_disalloawed** – boolean

**Returns**
whether this command is allowed to run.

**Return type**
boolean

**:raise CommandError if the command is not allowed and rise**
error flag is enabled.

**do**() → Tuple[ska_tango_base.commands.ResultCode, str]

Invoke the *reset* command on the Controller ComponentManager.

**Returns**
a tuple with the command result code and an informative message.

**AbortCommands**() → Tuple[ska_tango_base.commands.ResultCode, str]

Aborts all the scheduled and running tasks.

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

Bases: ResponseCommand

Class to abort all the scheduled and running commands

**do**() → Tuple[ska_tango_base.commands.ResultCode, str]

Invoke the *on* command on the Controller ComponentManager.

**Parameters**
**argin** – A list of TANGO Device fqdn to switch on.

**Returns**
a tuple with the command result code and an informative message.

**vlbiBeamsAddresses**

Device attribute for long running commands.

**longRunningCommandStatus**

Device attribute for long running commands.

**write_adminMode**(*value: ska_tango_base.control_model.AdminMode*) → None

Set the administration mode for the whole CSP element.

**Parameters**
**value** (*AdminMode*) – one of the administration mode value (ON-LINE, OFF-LINE, MAINTENANCE, NOT-FITTED, RESERVED).

**read_onCommandProgress**() → int

Return the progress for the execution of the On() CSP task.

**read_offCommandProgress**() → int

Return the progress for the execution of the Off() CSP task.

**read_standbyCommandProgress**() → int

Return the progress for the execution of the Standby() CSP task.

**read_onCmdDurationExpected**() → int

Return the value configured for the duration time expected to execute the *On* task.

**write_onCmdDurationExpected**(*value: int*) → None

Set the value configured for the duration time expected to execute the *On* task.

TODO: The input value should be compared with the sum of the duration times configured for each subordinate on-line sub-system (if any). The configured duration time for the command should be set equal the max

NOTE: not implemented

**Parameters**
**value** – the duration time expected

**read_offCmdDurationExpected**() → int

Return the value configured for the duration time expected to execute the *Off* task.

NOTE: not implemented

**Parameters**
**value** – the duration time expected

**write_offCmdDurationExpected**(*value*) → None

Set the value configured for the duration time expected to execute the *Off* task.

TODO: The input value should be compared with the sum of the duration times configured for each subordinate on-line sub-system (if any). The configured duration time for the command should be set equal the max

NOTE: not implemented

**Parameters**
**value** – the duration time expected

**read_standbyCmdDurationExpected**() → int

Return the value configured for the duration time expected to execute the *Standby* task.

NOTE: not implemented

**Parameters**
**value** – the duration time expected

**write_standbyCmdDurationExpected**(*value*) → None

Set the value configured for the duration time expected to execute the *Off* task.

TODO: The input value should be compared with the sum of the duration times configured for each subordinate on-line sub-system (if any). The configured duration time for the command should be set equal the max

NOTE: not implemented

**Parameters**
**value** – the duration time expected

**read_onCmdDurationMeasured**() → int

Return effective execution time for the CSP *On* task.

NOTE: not implemented

**read_offCmdDurationMeasured**() → int

Return effective execution time for the CSP *Off* task.

NOTE: not implemented

**read_standbyCmdDurationMeasured**() → int

Return effective execution time for the CSP *Standby* task.

NOTE: not implemented

**read_onCmdTimeoutExpired**() → bool

Return whether the On command timeout has expired.

NOTE: not implemented

**read_offCmdTimeoutExpired**() → bool

Return whether the On command timeout has expired.

NOTE: not implemented

**read_standbyCmdTimeoutExpired**() → bool

Return whether the On command timeout has expired.

NOTE: not implemented

**read_cspCbfState**() → tango.DevState

Return the cspCbfState attribute.

**read_cspPssState**() → tango.DevState

Return the cspPssState attribute.

**read_cspPstBeamsState**() → str

Return the Pst Beams State.

**read_cspCbfHealthState**() → tango.DevState

Return the cspCbfHealthState attribute.

**read_cspPssHealthState**() → tango.DevState

Return the cspPssHealthState attribute.

**read_cspPstBeamsHealthState**() → tango.DevState

Return the Pst Beams health state.

**read_cbfMasterAddress**() → str

Return the cbfMasterAddress attribute.

**read_pssMasterAddress**() → str

Return the pssMasterAddress attribute.

**read_pstBeamsAddresses**() → List[str]

Return the PstBeams FQDNs values.

**read_cspCbfAdminMode**() → ska_tango_base.control_model.AdminMode

Return the cspCbfAdminMode attribute.

**write_cspCbfAdminMode**(*value*) → ska_tango_base.control_model.AdminMode

Write attribute method.

Set the CBF sub-element *adminMode* attribute value.

**Parameters**

**value** – one of the administration mode value (ON-LINE, OFF-LINE, MAINTENANCE, NOT-FITTED, RESERVED).

**Returns**

None

**Raises**
tango.DevFailed when there is no DeviceProxy providing interface to the CBF sub-element Master, or an exception is caught in command execution.

**read_cspPssAdminMode**() → ska_tango_base.control_model.AdminMode

Return the cspPssAdminMode attribute.

**write_cspPssAdminMode**(*value*) → ska_tango_base.control_model.AdminMode

Set the cspPssAdminMode attribute.

NOTE: not implemented

**read_cspPstBeamsAdminMode**() → str

Return the cspPstAdminMode attribute.

**read_numOfDevCompletedTask**()

Return the numOfDevCompletedTask attribute.

NOTE: not implemented

**read_onCmdFailure**()

Return the onCmdFailure attribute.

NOTE: not implemented

**read_onFailureMessage**()

Return the onFailureMessage attribute.

NOTE: not implemented

**read_offCmdFailure**()

Return the offCmdFailure attribute.

NOTE: not implemented

**read_offFailureMessage**()

Return the offFailureMessage attribute.

NOTE: not implemented

**read_standbyCmdFailure**()

Return the standbyCmdFailure attribute.

NOTE: not implemented

**read_standbyFailureMessage**()

Return the standbyFailureMessage attribute.

NOTE: not implemented

**read_cspSubarrayAddresses**() → List[str]

Return the list with the CspSubarrays' addresses (FQDNs).

NOTE: not implemented

**read_listOfDevCompletedTask**()

Return the list of the devices (FQDNs) that have completed the last required task.

NOTE: not implemented

**read_reservedSearchBeamIDs**() → List[int]

Return the list with IDs of the SearchBeams that are reserved to a subarray.

*Note*: SearchBeams associated to the same processing node (GPU) must be assigned to the same subarray. If one of these beams is not assigned, it has to be reserved.

NOTE: not implemented

**read_searchBeamsAddresses**() → List[str]

Return a list with the SearchBeams Capabilities TANGO devices addresses (FQDNs).

NOTE: not implemented

**read_timingBeamsAddresses**() → List[str]

Return a list with the TimingBeams Capabilities TANGO devices addresses (FQDNs).

NOTE: not implemented

**read_vlbiBeamsAddresses**() → List[str]

Return a list with the VlbiBeams Capabilities TANGO devices addresses (FQDNs).

NOTE: not implemented

**read_commandResult**() → Tuple[str, ska_tango_base.commands.ResultCode]

Return the commandResult value. Implemented as a list of 2 strings:

- element 0: CSP task name
- element 1: ResultCode value

This attribute is used to signal the client the completion of the execution of a non-blocking task.

**read_isCommunicating**() → bool

Whether the TANGO device is communicating with the controlled component.

**read_longRunningCommandStatus**()

Read the status of the currently executing long running commands.

**Returns**
ID, status pairs of the currently executing commands

**read_longRunningCommandResult**()

Read the result of the completed long running command.

**Returns**
ID, ResultCode, result.

### 3.1.2 CspSubarray

**class** ska_csp_lmc_common.subarray_device.**CspSubarray**(**args: Any*, ***kwargs: Any*)

CSP subarray functionality is modeled via a TANGO CSP.LMC Common Class for the CSPSubarray TANGO Device.

**Device Properties**

**CspMaster**

- The TANGO address of the CspMaster.
- Type:'DevString'

**CbfSubarray**

- CBF sub-element sub-array TANGO device FQDN
- Type:'DevString'

**PssSubarray**

- PST sub-element sub-array TANGO device FQDN.
- Type:'DevString'

**PstBeams**

- PST sub-element PstBeams TANGO devices FQDNs
- Type:'DevVarStringArray'

**SubarrayProcModeCorrelation**

- CSP Subarray *Correlation Inherent Capability* TANGO device FQDN
- Type:'DevString'

**SubarrayProcModePss**

- CSP Subarray *Pss Inherent Capability* TANGO device FQDN
- Type:'DevString'

**SubarrayProcModePst**

- CSP Subarray *PST Inherent Capability* TANGO device FQDN
- Type:'DevString'

**SubarrayProcModeVlbi**

- CSP Subarray *VLBI Inherent Capability* TANGO device FQDN
- Type:'DevString'

`init_device()` → None

Override the Base Classes *init_device* method to change the asynchronous callback sub-model from pull to push sub-model.

`_init_state_model()` → None

Override the health, operational and observing State models:

current CSP.LMC implementation does no longer rely on the SKA State models and associated State Machine library.

`set_component_manager`(*cm_configuration: ComponentManagerConfiguration*) → CSPSubarrayComponentManager

Configure the ComponentManager for the CspSubarray TANGO device.

This method has to be specialized in Mid.CSP and Low.CSP.

**Parameters**
`cm_configuration` (`ComponentManagerConfiguration`) – A class with all the device properties accessible as attributes

`create_component_manager()` → CSPSubarrayComponentManager

Override the SKA BC method.

**Returns**
The CSP SubarrayComponentManager

**Return type**
:py:class`CSPSubarrayComponentManager`

**init_command_objects**() → None

Specialize the SKA Base Classes *init_command_objects* method to support new device commands.

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

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

**do**() → Tuple[ska_tango_base.commands.ResultCode, str]

Stateless hook for device initialization.

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

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

Class for the subarray On command.

The class inherits from the base ResponseCommand because current implementation does not rely on the SKA Operational State Model.

**is_allowed**(*raise_if_disallowed: bool = True*) → bool

Method invoked by the TANGO method *is_On_allowed()*. The *On* command on a CSP subarray is allowed only when the device is in OFF state.

**Parameters**
**raise_if_disallowed** – whether to raise an error or simply return False if the command is disallowed. Currently not used.

**Returns**
whether this command is allowed to run

**Raises**
**tango.DevFailed** – if the command is not allowed and *raise_if_disallowed* is True

**do**() → Tuple[ska_tango_base.commands.ResultCode, str]

Invoke the *on* command on the SubarrayComponentManager.

**Returns**
a tuple with the result code and the associated message.

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

Class for the subarray *Off* command.

The class inherits from the base ResponseCommand because current implementation does not rely on the SKA Operational State Model.

**is_allowed**(*raise_if_disallowed: bool = True*) → bool

Method invoked by the TANGO method *is_Off_allowed()*.

*Off* command is allowed from any state.

**Parameters**
**raise_if_disallowed** – whether to raise an error or simply return False if the command is disallowed. Currently not used.

**Returns**
whether this command is allowed to run

**do**() → Tuple[ska_tango_base.commands.ResultCode, str]

Invoke the *off* command on the SubarrayComponentManager.

**Returns**
a tuple with the result code and the associated message.

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

Class to handle the Configure task for a CSP Subarray.

The class inherits from the base ResponseCommand because current implementation does not rely on the SKA Operational State Model.

is_allowed(*raise_if_disallowed: bool = False*) → bool

Method invoked by the TANGO method *is_Configure_allowed()*. The *Configure* command on a CSP subarray is allowed only when the device is in ON state and the observing state is IDLE or READY.

**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

**Raises**
tango.DevFailed – if the command is not allowed and *raise_if_disallowed* is True

do(*argin: Dict*) → Tuple[ska_tango_base.commands.ResultCode, str]

Invoke the *configure* command on the SubarrayComponentManager.

**Parameters**
**argin** – Dictionary with the configuration for each CSP subordinate sub-system.

**Returns**
a tuple with the result code and the associated message.

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

Class for the subarray GoToIdle command.

The class inherits from the base ResponseCommand because current implementation does not rely on the SKA Observational State Model.

is_allowed() → bool

Method invoked by the TANGO method *is_GoToIdle_allowed()*.

The *GoToIdle* command on a CSP subarray is allowed only when the device is in READY state.

**Returns**
`True` if the command is allowed

**Raises**
tango.DevFailed – if the command is not allowed and *raise_if_disallowed* is True

do() → Tuple[ska_tango_base.commands.ResultCode, str]

Invoke the GoToIdle command on the SubarrayComponentManager.

**Returns**
a tuple with the result code and the associated message.

is_GoToIdle_allowed() → bool

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

**Returns**
`True` if the command is allowed

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

Class for the subarray Abort command.

The class inherits from the base ResponseCommand because current implementation does not rely on the SKA Operational State Model.

is_allowed(*raise_if_disallowed: bool = True*) → bool

*Abort* command can be issued on a CSP subarray only when:

- the device is in ON state and
- **the device obsState is one of the following observing state:**
  IDLE, CONFIGURING, READY, SCANNING, RESETTING

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

**do**() → Tuple[ska_tango_base.commands.ResultCode, str]

Invoke the *abort* command on the SubarrayComponentManager.
**Returns**
a tuple with the result code and the associated message.

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

Class for the subarray ObsReset command.

The class inherits from the base ResponseCommand because current implementation does not rely on the SKA Operational State Model.

**is_allowed**(*raise_if_disallowed: bool = True*) → bool

*ObsReset* command can be issued on a CSP subarray only when:
- the device is in ON state and
- the device obsState is FAULT or ABORTED

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

**do**() → Tuple[ska_tango_base.commands.ResultCode, str]

Invoke the *obsreset* command on the SubarrayComponentManager.
**Returns**
a tuple with the result code and the associated message.

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

Class for handling the CSP Subarray Restart command.

**is_allowed**(*raise_if_disallowed: bool = True*) → bool

*Restart* command can be issued on a CSP subarray only when:
- the device is in ON state and
- the device obsState is one of the following observing state: ABORTED, FAULT

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

**do**() → Tuple[ska_tango_base.commands.ResultCode, str]

Invoke the *restart* command on the SubarrayComponentManager.

**Returns**

a tuple with the result code and the associated message.

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

Class for handling the CSP Subarray AssignResources command.

The class inherits from the base ResponseCommand because current implementation does not rely on the SKA Operational State Model.

**is_allowed**(*raise_if_disallowed: bool = True*) → bool

Assign Resources command can be issued on a CSP subarray only when:

- the device is in ON state and
- the device obsSTate is one of the following observing state: IDLE

**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

**Raises**

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

**do**(*argin: Dict*) → Tuple[ska_tango_base.commands.ResultCode, str]

Invoke the *assignresources* command on the SubarrayComponentManager.

**Parameters**

**argin** – a dictionary with resources to assign to the CSP Subarray.

**Returns**

a tuple with the result code and the associated message.

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

Class for handling the CSP Subarray ReleaseResources command.

The class inherits from the base ResponseCommand because current implementation does not rely on the SKA Operational State Model.

**is_allowed**(*raise_if_disallowed: bool = True*) → bool

Release Resources command can be issued on a CSP subarray only when:

- the device is in ON state and
- the device obsSTate is one of the following observing state:IDLE

**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

**Raises**

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

**do**(*argin: Dict*) → Tuple[ska_tango_base.commands.ResultCode, str]

Invoke the *releaseresources* command on the SubarrayComponentManager.

**Parameters**

**argin** – a dictionary with resources to remove from the CspSubarray.

**Returns**

a tuple with the result code and the associated message.

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

Class for handling the CSP Subarray ReleaseAllResources command.

The class inherits from the base ResponseCommand because current implementation does not rely on the SKA Operational State Model.

**is_allowed**(*raise_if_disallowed: bool = True*) → bool

Release All Resources command can be issued on a CSP subarray only when:

- the device is in ON state and
- the device obsState is one of the following observing state:IDLE

**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

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

**do**() → Tuple[ska_tango_base.commands.ResultCode, str]

Invoke the *releaseallresources* command on the SubarrayComponentManager.

**Returns**
a tuple with the result code and the associated message.

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

Class for the subarray Scan command.

The class inherits from the base ResponseCommand because current implementation does not rely on the SKA Operational State Model.

**is_allowed**(*raise_if_disallowed: bool = True*) → bool

Scan command can be issued on a CSP subarray only when:

- the device is in ON state and
- the device obsState is in READY

**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

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

**do**(*argin: Dict*) → Tuple[ska_tango_base.commands.ResultCode, str]

Invoke the *scan* command on the SubarrayComponentManager.

**Parameters**
**argin** – A dictionary the information needed to start the scan.

**Returns**
a tuple with the result code and the associated message.

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

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

Class for the Reset command.

The class inherits from the base ResponseCommand because current implementation does not rely on the SKA Operational State Model.

**is_allowed**(*raise_if_disallowed: bool = True*) → bool

Method invoked by the TANGO method is_Reset_allowed() to determine whether the Reset command can be invoked on the device. The Reset command is allowed from any state except INIT: in this way, even if the CSP Subarray is not in FAULT, it is possible to reset the faulty condition of a CSP sub-system.

**Parameters**
**raise_if_disallowed** – whether to raise an error or simply return False if the command is disallowed. Currently not used.

**Returns**
whether this command is allowed to run

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

**do**() → Tuple[ska_tango_base.commands.ResultCode, str]

Invoke the *reset* command on the SubarraycomponentManager.

**Returns**
A tuple with the command result code and an informative message

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

Class for the subarray EndScan command.

The class inherits from the base ResponseCommand because current implementation does not rely on the SKA Operational State Model.

**is_allowed**(*raise_if_disallowed: bool = True*) → bool

*EndScan* command can be issued on a CSP subarray only when:
- the device is in ON state and
- the device obsSTate is one of the following observing state: SCANNING

**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

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

**do**() → Tuple[ska_tango_base.commands.ResultCode, str]

Invoke the *endscan* command on the SubarraycomponentManager.

**Returns**
A tuple with the command result code and an informative message

**update_subarray_state**(*value: tango.DevState*) → None

Update the CSP Subarray state to the value returned by the EventManager that aggregates the state of the CSP Subarray subordinate components. Update is performed only when no CSP command is running.

The method re-calculate the CSP Subarray healthState as result of the State change in one or more CSP Subarray sub-systems.

**Parameters**
**value** – the CSP Subarray state calculated aggregating the state of the CSP Subarray sub-systems

**update_subarray_health_state**(*value: ska_tango_base.control_model.HealthState*) → None

Method invoked when a change happens in the CSP healthState.

**Parameters**
**value** – the updated value via events

**update_subarray_obs_state**(*value: ska_tango_base.control_model.ObsState*) → None

Update the CSP Subarray obsState to the value returned by the EventManager that aggregates the obsState of the CSP Subarray subordinate components. Update is performed only when no CSP command is running.

**Parameters**
**value** – the updated value via events

**update_property**(*property_name: str*, *property_value: Any*) → None

General method invoked by the SubarrayComponentManager to push an event on a device attribute properly configured to push events from the device.

**Parameters**

- **property_name** – the TANGO attribute name
- **property_value** – the attribute value

**isCommunicating**

Device attribute for long running commands.

**longRunningCommandStatus**

Device attribute for long running commands.

**always_executed_hook**()

Method always executed before any TANGO command is executed.

**delete_device**() → None

Hook to delete resources allocated in init_device.

This method allows for any memory or other resources allocated in the init_device method to be released. This method is called by the device destructor and by the device Init command.

**set_state**(*state*)

Put the state change on internal queue.

**set_status**(*status*)

Put the state change on internal queue.

**push_change_event**(*name*, *value=None*)

Put the change event on internal queue.

**PushChanges**()

Periodic method to push events.

Get the events from a queue and pushes them.

**write_adminMode**(*value: ska_tango_base.control_model.AdminMode*) → None

Set the administration mode for the whole CSP element.

**Parameters**
**value** (*AdminMode*) – one of the administration mode value (ON-LINE, OFF-LINE, MAINTENANCE, NOT-FITTED, RESERVED).

**read_scanID**() → int

Return the scanID attribute.

**Returns**
the Subarray scanID

**write_scanID**(*value: int*) → None

Set the scanID attribute.

:param The Subarray scanID

**read_procModeCorrelationAddr**() → str

Return the procModeCorrelationAddr attribute.

**Returns**
CSP Subarray *Correlation Inherent Capability* TANGO device FQDN

**read_procModePssAddr**() → str

Return the procModePssAddr attribute.

**Returns**
The CSP sub-array PSS Inherent Capability FQDN.

**read_procModePstAddr**() → str

Return the procModePstAddr attribute.

**Returns**
The CSP sub-array PST Inherent Capability FQDN.

**read_procModeVlbiAddr**() → str

Return the procModeVlbiAddr attribute.

**Returns**
The CSP sub-array VLBI Inherent Capability FQDN.

**read_cbfSubarrayState**() → tango.DevState

Return the cbfSubarrayState attribute.

**Returns**
The correspndant CBF Subarray Operational State

**read_pssSubarrayState**() → tango.DevState

Return the pssSubarrayState attribute.

**Returns**
The correspndant PSS Subarray Operational State

**read_cbfSubarrayHealthState**() → ska_tango_base.control_model.HealthState

Return the cbfSubarrayHealthState attribute.

**Returns**
The correspndant CBF Subarray Health State

**read_pssSubarrayHealthState**() → ska_tango_base.control_model.HealthState

Return the pssSubarrayHealthState attribute.

**Returns**
The correspndant PSS Subarray Health State

**read_cbfSubarrayAdminMode**() → ska_tango_base.control_model.AdminMode

Return the cbfSubarrayAdminMode attribute.

**Returns**
The correspndant CBF Subarray Administation Mode

**read_pssSubarrayAdminMode**() → ska_tango_base.control_model.AdminMode

Return the pssSubarrayAdminMode attribute.

**Returns**
The correspndant PSS Subarray Administation Mode

**read_cbfSubarrayObsState**() → ska_tango_base.control_model.ObsState

Return the cbfSubarrayObsState attribute.

**Returns**
The correspndant CBF Subarray Observational State

**read_pssSubarrayObsState**() → ska_tango_base.control_model.ObsState

Return the pssSubarrayObsState attribute.

**Returns**
The correspndant PSS Subarray Observational State

**read_pssSubarrayAddr**() → str

Return the pssSubarrayAddr attribute.

**Returns**
The PSS sub-element sub-array FQDN.

**read_cbfSubarrayAddr**() → str

Return the cbfSubarrayAddr attribute.

**Returns**
The CBF sub-element sub-array FQDN.

**read_validScanConfiguration**() → str

Return the validScanConfiguration attribute.

**Returns**
the JSON string containing the last valid scan configuration.

**read_addSearchBeamsDurationExpected**() → int

Return the addSearchBeamsDurationExpected attribute.

**Returns**
The duration expected (in sec) for the AddSearchBeams command.

NOTE: not implemented

**write_addSearchBeamsDurationExpected**(*value*) → None

Set the addSearchBeamsDurationExpected attribute.

**Param**
The duration expected (in sec) for the AddSearchBeams command.

NOTE: not implemented

**read_remSearchBeamsDurationExpected**() → int

Return the remSearchBeamsDurationExpected attribute.

**Returns**
The duration expected (in sec) for the RemoveSearchBeams command.

NOTE: not implemented

write_remSearchBeamsDurationExpected(*value: int*) → None

Set the remSearchBeamsDurationExpected attribute.

**Param**
The duration expected (in sec) for the RemoveSearchBeams command.

NOTE: not implemented

read_addSearchBeamsDurationMeasured() → int

Return the addSearchBeamsDurationMeasured attribute.

**Returns**
The duration measured (in sec) for the AddSearchBeams command.

NOTE: not implemented

read_remSearchBeamsDurationMeasured() → int

Return the remSearchBeamsDurationMeasured attribute.

**Returns**
The duration measured (in sec) for the RemoveSearchBeams command.

NOTE: not implemented

read_addTimingBeamsDurationExpected() → int

Return the addTimingBeamsDurationExpected attribute.

**Return**
The duration expected (in sec) for the AddTimingBeams command.

NOTE: not implemented

write_addTimingBeamsDurationExpected(*value: int*) → None

Set the addTimingBeamsDurationExpected attribute.

**Param**
The duration expected (in sec) for the AddTimingBeams command.

NOTE: not implemented

read_remTimingBeamsDurationExpected() → int

Return the remTimingBeamsDurationExpected attribute.

**Returns**
The duration expected (in sec) for the RemoveTimingBeams command.

NOTE: not implemented

write_remTimingBeamsDurationExpected(*value: int*) → None

Set the remTimingBeamsDurationExpected attribute.

**Param**
The duration expected (in sec) for the RemoveTimingBeams command.

NOTE: not implemented

read_addTimingBeamsDurationMeasured() → int

Return the addTimingBeamDurationMeasured attribute.

**Returns**
The duration measured (in sec) for the AddTimingBeams command.

NOTE: not implemented

**read_remTimingBeamsDurationMeasured**() → int

Return the remTimingBeamsDurationMeasured attribute.

**Returns**
The duration measured (in sec) for the RemoveTimingBeams command.

NOTE: not implemented

**read_addResourcesCmdProgress**() → int

Return the assResourcesCmdProgress attribute.

**Returns**
The duration expected (in sec) for the Assign Resources command.

NOTE: not implemented

**read_assignresourcesDurationExpected**() → int

Return the assignresourcesDurationExpected attribute.

**Returns**
The duration expected (in sec) for the Assign Resources command.

NOTE: not implemented

**write_assignresourcesDurationExpected**(*value: int*) → None

Set the assignresourcesDurationExpected attribute.

**Param**
The duration expected (in sec) for the Assign Resources command.

NOTE: not implemented

**read_assignresourcesDurationMeasured**() → int

Return the assignresourcesDurationMeasured attribute.

**Returns**
The duration measured (in sec) for the Assign Resources command.

NOTE: not implemented

**read_goToIdleDurationExpected**() → int

Return the assignresourcesDurationExpected attribute.

**Returns**
The duration expected (in sec) for the GoToIdle command.

NOTE: not implemented

**write_goToIdleDurationExpected**(*value: int*) → None

Set the goToIdleDurationExpected attribute.

**Param**
The duration expected (in sec) for the GoToIdle command.

NOTE: not implemented

**read_goToIdleDurationMeasured**() → int

Return the goToIdleDurationMeasured attribute.

**Returns**
The duration measured (in sec) for the GoToIdle command.

NOTE: not implemented

read_endScanDurationExpected() → int

Return the endScanDurationExpected attribute.

**Returns**
The duration expected (in sec) for the EndScan command.

NOTE: not implemented

read_endScanDurationMeasured() → int

Return the endScanDurationMeasured attribute.

**Returns**
The duration measured (in sec) for the EndScan command.

NOTE: not implemented

read_removeResourcesCmdProgress() → int

Return the removeResourcesCmdProgress attribute.

**Returns**
The progress percentage for the Remove Resources command.

NOTE: not implemented

read_assignresourcesCmdProgress() → int

Return the assignresourcesCmdProgress attribute.

**Returns**
The progress percentage for the Assign Resources command.

NOTE: not implemented

read_goToIdleCmdProgress() → int

Return the goToIdleCmdProgress attribute.

**Returns**
The progress percentage for the GoToIdle command.

NOTE: not implemented

read_endScanCmdProgress() → int

Return the endScanCmdProgress attribute.

**Returns**
The progress percentage for the EndScan command.

NOTE: not implemented

read_reservedSearchBeamNum() → int

Return the reservedSearchBeamNum attribute.

**Returns**
Number of SearchBeam IDs reserved for the CSP sub-array

NOTE: not implemented

read_numOfDevCompletedTask() → int

Return the numOfDevCompletedTask attribute.

**Returns**
Number of devices that completed the task

NOTE: not implemented

**read_assignedSearchBeamIDs**() → Tuple[int]

Return the assignedSearchBeamIDs attribute.

**Returns**
List of assigned Search Beams

NOTE: not implemented

**read_reservedSearchBeamIDs**() → Tuple[int]

Return the reservedSearchBeamIDs attribute.

**Returns**
List of SearchBeam IDs reserved for the CSP sub-array

NOTE: not implemented

**read_assignedTimingBeamIDs**() → Tuple[int]

Return the assignedTimingBeamIDs attribute.

**Returns**
List of TimingBeam IDs assigned to the CSP sub-array

**read_assignedTimingBeams**() → None

Return the assignedTimingBeamIDs attribute.

**Returns**
Write the TimingBeam IDams assigned to the CSP sub-array

NOTE: not implemented

**write_assignedTimingBeams**(*assigned_timing_beams*) → None

Return the assignedTimingBeamIDs attribute.

**Returns**
Write the string with TimingBeam IDs assigned to the

CSP sub-array. This attribute is memorized so that at device restart it can be recovered. The correspondent Spectrum attribute can't be configured as memorized attribute. Temporary solution to use until the introduction of the PST Processing Mode Capability device.

**read_assignedVlbiBeamIDs**() → Tuple[int]

Return the assignedVlbiBeamIDs attribute.

**Returns**
List of VlbiBeam IDs assigned to the CSP sub-array

NOTE: not implemented

**read_assignedSearchBeamsState**() → Tuple[tango.DevState]

Return the assignedSearchBeamsState attribute.

**Returns**
State of the assigned SearchBeams

NOTE: not implemented

**read_assignedTimingBeamsState**() → Tuple[tango.DevState]

Return the assignedTimingBeamsState attribute.

**Returns**
State of the assigned TimingBeams

NOTE: not implemented

**read_assignedVlbiBeamsState**() → Tuple[tango.DevState]

Return the assignedVlbiBeamsState attribute.

**Returns**
State of the assigned VlbiBeams

NOTE: not implemented

**read_assignedSearchBeamsHealthState**() → Tuple[int]

Return the assignedSearchBeamsHealthState attribute.

**Returns**
HealthState of the assigned SearchBeams

NOTE: not implemented

**read_assignedTimingBeamsHealthState**() → Tuple[int]

Return the assignedTimingBeamsHealthState attribute.

**Returns**
HealthState of the assigned TimingBeams

NOTE: not implemented

**read_assignedVlbiBeamsHealthState**() → Tuple[int]

Return the assignedVlbiBeamsHealthState attribute.

**Returns**
HealthState of the assigned VlbiBeams

NOTE: not implemented

**read_assignedSearchBeamsObsState**() → Tuple[int]

Return the assignedSearchBeamsObsState attribute.

**Returns**
ObsState of the assigned SearchBeams

NOTE: not implemented

**read_assignedTimingBeamsObsState**() → Tuple[int]

Return the assignedTimingBeamsObsState attribute.

**Returns**
ObsState of the assigned TimingBeams

NOTE: not implemented

**read_assignedVlbiBeamsObsState**() → Tuple[int]

Return the assignedVlbiBeamsObsState attribute.

**Returns**
ObsState of the assigned VlbiBeams

NOTE: not implemented

**read_assignedSearchBeamsAdminMode**() → Tuple[int]

Return the assignedSearchBeamsAdminMode attribute.

**Returns**
AdminMode of the assigned SearchBeams

NOTE: not implemented

**read_assignedTimingBeamsAdminMode**() → Tuple[int]

Return the assignedTimingBeamsAdminMode attribute.

**Returns**
AdminMode of the assigned TimingBeams

NOTE: not implemented

**read_assignedVlbiBeamsAdminMode**() → Tuple[int]

Return the assignedVlbiBeamsAdminMode attribute.

**Returns**
AdminMode of the assigned VlbiBeams

NOTE: not implemented

**read_pstBeamsAddr**() → str

Return the pstBeamsAddr attribute.

**Returns**
PST sub-element PstBeam TANGO device FQDNs.

**read_pstBeamsState**() → str

Return State value for the PST Beams.

**Returns**
A JSON formatted string with the state of the pst Beams assigned to the subarray.

**read_pstBeamsObsState**() → str

Return the obsState values for the PST Beams.

**Returns**
A JSON formatted string with the observing state of the PST Beams assigned to the subarray.

**read_pstBeamsHealthState**() → str

Return the health state values of the PST Beams.

**Returns**
A JSON formatted string with the health state of the PST beams assigned to the subarray.

**read_pstBeamsAdminMode**() → str

Return the administration Mode of the PST Beams.

**Returns**
A JSON formatted string with the administration mode of the PST Beams assigned to the subarray.

**read_scanCmdProgress**() → int

Return the scanCmdProgress attribute.

**Returns**
The progress percentage for the Scan command.

NOTE: not implemented

**read_timeoutExpiredFlag**() → bool

Return the timeoutExpiredFlag attribute.

**Returns**
The timeout flag for a CspSubarray command.

**read_failureRaisedFlag**() → bool

Return the failureRaisedFlag attribute.

**Returns**
The failure flag for a CspSubarray command

**read_isCmdInProgress**() → bool

Return the isCmdInProgress attribute.

NOTE: not implemented

**read_cmdFailureMessage**() → str

Return the cmdFailureMessage attribute.

**Returns**
The failure message on command execution

**read_pstOutputLink**() → str

Return the pstOutputLink attribute.

**Returns**
The output link for PST products.

NOTE: not implemented

**read_listOfDevCompletedTask**() → List[str]

Return the listOfDevCompletedTask attribute.

**Returns**
List of devices that completed the task

NOTE: not implemented

**read_configurationID**() → int

Return the configurationID attribute.

**Returns**
The configuration identifier

**read_commandResult**() → Tuple[ska_tango_base.commands.ResultCode, str]

Return the configurationID attribute.

**Returns**
the name and the result_code of latest command execution

**read_isCommunicating**() → bool

Whether the TANGO device is communicating with the controlled component.

**Returns**
Whether the device is communicating with the component under control.

**read_longRunningCommandStatus**()

Read the status of the currently executing long running commands.

**Returns**
ID, status pairs of the currently executing commands

**read_longRunningCommandResult**()

Read the result of the completed long running command.

**Returns**
ID, ResultCode, result.

## 3.2 CSP.LMC modules API

### 3.2.1 Manager subpackage

#### Controller Component Manager

**class** ska_csp_lmc_common.manager.**CSPControllerComponentManager**(**args: Any*, ***kwargs: Any*)

Bases: CSPBaseComponentManager

Component Manager for the Csp Controller.

**__init__**(*op_state_model: ControllerOpStateModel*, *health_model: ControllerHealthModel*, *properties: ComponentManagerConfiguration*, *update_device_property_cbk: Optional[Callable] = None*, *logger: Optional[Logger] = None*) → None

The Component Manager for the CSP Controller device.

**Parameters**

- **op_state_model** – The Operational State Model to evaluate the CSP Controller device state
- **health_model** – The Health State Model to evaluate the CSP Controller health state.
- **properties** – A class instance whose properties are the CSP Controller device properties.
- **update_device_property_cbk** – The CSP Controller method invoked to update the properties. Defaults to None.
- **logger** – The device or python logger if default is None.

**_controller_factory**(*logger: Logger*) → ControllerComponentFactory

Configure the Factory to create the sub-systems components.

This class is specialized in Mid.CSP and Low.CSP.

**Parameters**
**logger** – the logger for this instance.

**Returns**
A factory object to create CSP Controller sub-systems observing components.

**_add_subsystem_components**() → None

Instantiate a Python class for each CSP sub-system component.

Each sub-system component works as an adapter and cache for the associated CSP sub-system. The sub-system component reports the information about one CSP subordinate device and provides the methods or calls to operate on the associate sub-system TANGO device.

On failure, the state and health state of the CSP Controller is forced to FAULT/FAILED.

**_populate_subsystems_list**(*argin: List[str]*) → List[Component]

Populate a list with the Csp Subsystems components that the controller has to operate on. If argin is an empty list, the CSP Controller operates on all the online Subsystems. If any fqdn is invalid return an empty list.

**Parameters**
**argin** – List of fqdn of controller's subsystems to operate

**Return argout**
List of components to operate on

**_on**(*task_name: str*, *command_id: str*, *argin: List[Component]*, *task_callback: Optional[Callable] = None*, *task_abort_event: Optional[Event] = None*) → None

Method submitted into thread executor for on command

**Parameters**

- **task_name** – the name of the task
- **argin** – list with FQDNs of the CSP Sub-systems Controllers to operate on. If the list is empty operated on all of them.

**Param**
command_id: unique task identifier

**Param**
task_callback: method called to update the task result related attributes

**Param**
task_abort_event: flag to signal an abort request pending

**_off**(*task_name*, *command_id: str*, *argin: List[Component]*, *task_callback: Optional[Callable] = None*, *task_abort_event: Optional[Event] = None*) → None

Method submitted into thread executor for off command

**Parameters**

- **task_name** – the name of the task
- **argin** – list with FQDNs of the CSP Sub-systems Controllers to operate on. If the list is empty operated on all of them.

**Param**
command_id: unique task identifier

**Param**
task_callback: method called to update the task result related attributes

**Param**
task_abort_event: flag to signal an abort request pending

**_standby**(*task_name: str*, *command_id: str*, *argin: List[Component]*, *task_callback: Optional[Callable] = None*, *task_abort_event: Optional[Event] = None*) → None

Method submitted into thread executor for standby command

**Parameters**

- **task_name** – the name of the task
- **argin** – list with FQDNs of the CSP Sub-systems Controllers to operate on. If the list is empty operated on all of them.

**Param**
command_id: unique task identifier

**Param**
task_callback: method called to update the task result related attributes

**Param**
task_abort_event: flag to signal an abort request pending

**init**() → None

Method to initialize the Controller ComponentManager. Instantiate:

- the command factory

- the Component factory
- the event manager
- a Component object for each CSP sub-system defined in the CSP Subarray Device Properties.

**start_communicating**() → None

Method to start the monitoring of the CSP subordinate devices, It is invoked when the adminMode of the device is ONLINE or MAINTENANCE.

Establish the connection with the CSP subordinate sub-systems Controller devices and with the CSP Subarrays. On success, the main sub-systems attributes are subscribed for events.

**stop_communicating**()

Method invoked when the CSP subarray is set OFFLINE. CSP Controller stop monitoring the subordinate sub-systems. Events on attributes are un-subscribed and proxies with sub-system devices are invalidated.

**TODO: Consider to maintain the proxy and subscription points active**
blocking only the aggregation and reporting of the events to the CSP device. In DISABLE state the main operations are prohibited and only 'emergency' actions could be allowed.

**on**(*argin: List[Component]*, *command_id: str*) → Tuple[ska_tango_base.commands.ResultCode, str]

Turn the CSP on. This operation is done powering on one sub-system after the other in an ordered way.

Powering sequence is handled via a semaphore acquired by the component executing the command.

**Parameters**
**argin** – list with FQDNs of the CSP Sub-systems Controllers to operate on. If the list is empty operated on all of them.

**Param**
command_id: unique task identifier

**Returns**
a tuple with *ResultCode* and an informative message.

**off**(*argin: List[Component]*, *command_id: str*) → Tuple[ska_tango_base.commands.ResultCode, str]

Turn the CSP off.

**Parameters**
**argin** – list with FQDNs of the CSP Sub-systems Controllers to operate on. If the list is empty operated on all of them.

**Param**
command_id: unique task identifier

**Returns**
a tuple with *ResultCode* and an informative message.

**standby**(*argin: List[Component]*, *command_id: str*) → Tuple[ska_tango_base.commands.ResultCode, str]

Send the component to low-power mode. This operation is done on one sub-system after the other in an ordered way. Powering sequence is handled via a semaphore acquired by the component executing the command.

**Parameters**
**argin** – list with FQDNs of the CSP Sub-systems Controllers to operate on. If the list is empty operated on all of them.

CSP Sub-system controllers are put in low-power mode, while the subarrays are switched off (observations not available in low power mode)

**Param**
command_id: unique task identifier

**Returns**
a tuple with *ResultCode* and an informative message.

### Subarray Component Manager

**class** ska_csp_lmc_common.manager.**CSPSubarrayComponentManager**(**args: Any*, ***kwargs: Any*)

Bases: CSPBaseComponentManager

Component Manager for the Csp Subarray.

**__init__**(*health_model: SubarrayHealthModel*, *op_state_model: SubarrayOpStateModel*, *obs_state_model: SubarrayObsStateModel*, *properties: ComponentManagerConfiguration*, *update_device_property_cbk: Optional[Callable] = None*, *logger: Optional[Logger] = None*) → None

Instantiate the Component Manager for the Csp Subarray.

**Parameters**

- **health_model** – The Health State Model to evaluate the CSP Subarray health state.
- **op_state_model** – The Operational State Model to evaluate the CSP Subarray device operational state.
- **obs_state_model** – The Observing State Model to evaluate the CSP Subarray device observing state.
- **properties** – A class instance whose properties are the CSP Subarray TANGO Device Properties.
- **update_device_property_cbk** – The CSP Subarray method invoked to update the properties. Defaults to None.
- **logger** – The device or python logger if default is None.

**property sub_id:** int

Return the subarray identification number.

**Returns**
The identification number of the CSP Subarray handled by the manager.

**property assigned_timing_beams_ids:** List[int]

Return the list of assigned timing beams IDs.

**property assigned_timing_beams_state:** List[int]

Return the state of the assigned timing beams

**property assigned_timing_beams_health_state:** List[int]

Return the state of the assigned timing beams

**_observing_factory**(*logger: Logger*) → ObservingComponentFactory

Configure the Factory to create the CSP sub-systems components.

This class is specialized in Mid.CSP and Low.CSP.

**Parameters**
**logger** – the logger for this instance.

**Returns**
A factory object to create CSP Subarray sub-systems observing components.

**_json_configuration_parser**()

Configure the class that handles the parsing of the input JSON files sent as argument of AssignResources, Configure and Scan.

This class is specialized in Mid.CSP and Low.CSP.

**Returns**
A reference to the class used to parse the subarray configuration JSON files.

**_add_subsystems_components**() → None

Instantiate a Python class for each CSP sub-system subordinate device.

Each sub-system component works as an adapter and cache for the associated CSP sub-system and provides the methods or calls to operate on the associate sub-system TANGO device.

On failure, the state and health state of the CSP Subarray is forced to FAULT/FAILED.

**_update_command_result**(*command_name: str*, *result_code: ska_tango_base.commands.ResultCode*, *command_id=None*, *task_status: Optional[TaskStatus] = None*, *result_msg: Optional[str] = None*) → None

Update the *commandResult* TANGO device property invoking the registered device callback, if any.

**Parameters**

- **command_name** – the name of the CSP command invoked on the Subarray.
- **result_code** – the command *ResultCode* value.

**_task_submitter**(*task_name: str*, *command_id: Optional[str] = None*, *argin: Optional[Any] = None*) → Tuple[ska_tango_base.commands.ResultCode, str]

Common method to sumbit a command

**Param**
task_name: the name of the task to be submitted

**Param**
command_id: unique task identifier

**Param**
argin: input argument of the task

**Returns**
a tuple with *ResultCode* and the command_id

**_on**(*task_name: str*, *command_id: str*, *task_callback: Optional[Callable] = None*, *task_abort_event: Optional[Event] = None*) → None

Method submitted into thread executor for on command

**Parameters**
**task_name** – the name of the task

**Param**
command_id: unique task identifier

**Param**
task_callback: method called to update the task result related attributes

**Param**
task_abort_event: flag to signal an abort request pending

**_off**(*task_name: str*, *command_id: str*, *task_callback: Optional[Callable] = None*, *task_abort_event: Optional[Event] = None*) → None

Method submitted into thread executor for off command

**Parameters**
**task_name** – the name of the task

**Param**
command_id: unique task identifier

**Param**
task_callback: method called to update the task result related attributes

**Param**
task_abort_event: flag to signal an abort request pending

**_pre_cbf_configure**()

Callback method invoked at CBF configuration.

If other processing modes are active besides IMAGING, CBF has to collect information about PST and/od PSS beams addresses to send beamforming data.

**Returns**
a dictionary with the PstBeams addresses

**configure_pst_beams**(*macro_input_list: List*, *resources_to_send: Dict*) → List

Configure PST beams when PST processing mode is active.

**configure_cbf**(*macro_input_list*, *resources_to_send*)

Configure CBF subarray.

**configure_pss**(*observer*, *resources_to_send: Dict*)

Configure PSS subarray when PSS processing mode is active.

**_configure**(*task_name: str*, *command_id: str*, *argin: dict*, *task_callback: Optional[Callable] = None*, *task_abort_event: Optional[Event] = None*)

Method submitted into thread executor for configure command.@

**Parameters**

- **task_name** – the name of the task
- **argin** – configuration input for Subarray

**Param**
command_id: unique task identifier

**Param**
task_callback: method called to update the task result related attributes

**Param**
task_abort_event: flag to signal an abort request pending

**_gotoidle**(*task_name: str*, *command_id: str*, *task_callback: Optional[Callable] = None*, *task_abort_event: Optional[Event] = None*) → None

Method submitted into thread executor for gotoidle command

**Parameters**
**task_name** – the name of the task

**Param**
command_id: unique task identifier

**Param**
task_callback: method called to update the task result related attributes

**Param**
task_abort_event: flag to signal an abort request pending

**_abort**(*task_name: str*, *command_id: str*, *task_callback: Optional[Callable] = None*, *task_abort_event: Optional[Event] = None*) → None

Method submitted into thread executor for abort command :param task_name: the name of the task :param: command_id: unique task identifier :param: task_callback: method called to update the task result

related attributes

**Param**
task_abort_event: flag to signal an abort request pending

**_obsreset**(*task_name: str*, *command_id: str*, *task_callback: Optional[Callable] = None*, *task_abort_event: Optional[Event] = None*) → None

Method submitted into thread executor for obsreset command

**Parameters**
**task_name** – the name of the task

**Param**
command_id: unique task identifier

**Param**
task_callback: method called to update the task result related attributes

**Param**
task_abort_event: flag to signal an abort request pending

**_restart**(*task_name: str*, *command_id: str*, *task_callback: Optional[Callable] = None*, *task_abort_event: Optional[Event] = None*) → None

Method submitted into thread executor for restart command

**Parameters**
**task_name** – the name of the task

**Param**
command_id: unique task identifier

**Param**
task_callback: method called to update the task result related attributes

**Param**
task_abort_event: flag to signal an abort request pending

**_assignresources**(*task_name: str*, *command_id: str*, *argin: dict*, *task_callback: Optional[Callable] = None*, *task_abort_event: Optional[Event] = None*)

Method submitted into thread executor for assignresources command

**Parameters**

- **task_name** – the name of the task
- **argin** – resources to be assigned to Subarray

**Param**
command_id: unique task identifier

**Param**
task_callback: method called to update the task result related attributes

**Param**
task_abort_event: flag to signal an abort request pending

**_release_completed**()

**_releaseresources**(*task_name: str*, *command_id: str*, *argin: dict*, *task_callback: Optional[Callable] = None*, *task_abort_event: Optional[Event] = None*)

Method submitted into thread executor for releaseresources command

**Parameters**

- **task_name** – the name of the task
- **argin** – resources to be assigned to Subarray

**Param**
command_id: unique task identifier

**Param**
task_callback: method called to update the task result related attributes

**Param**
task_abort_event: flag to signal an abort request pending

**_releaseallresources**(*task_name: str*, *command_id: str*, *task_callback: Optional[Callable] = None*, *task_abort_event: Optional[Event] = None*)

Method submitted into thread executor for releaseallresources command

**Parameters**

- **task_name** – the name of the task
- **argin** – resources to be assigned to Subarray

**Param**
command_id: unique task identifier

**Param**
task_callback: method called to update the task result related attributes

**Param**
task_abort_event: flag to signal an abort request pending

**_scan**(*task_name: str*, *command_id: str*, *argin: str*, *task_callback: Optional[Callable] = None*, *task_abort_event: Optional[Event] = None*)

Method submitted into thread executor for scan command

**Parameters**

- **task_name** – the name of the task
- **argin** – resources to be assigned to Subarray

**Param**
command_id: unique task identifier

**Param**
task_callback: method called to update the task result related attributes

**Param**
task_abort_event: flag to signal an abort request pending

**_endscan**(*task_name: str*, *command_id: str*, *task_callback: Optional[Callable] = None*, *task_abort_event: Optional[Event] = None*)

Method submitted into thread executor for endscan command

**Parameters**

- **task_name** – the name of the task
- **argin** – resources to be assigned to Subarray

**Param**
command_id: unique task identifier

**Param**
task_callback: method called to update the task result related attributes

**Param**
task_abort_event: flag to signal an abort request pending

**init**()

Initialize the Subarray ComponentManager. Instantiate:

- the command factory
- the json parser for the configuration files
- the event manager
- an ObservingComponent object for each CSP sub-system defined in the CSP Subarray Device Properties.

**start_communicating**()

Method to start the monitoring of the CSP subordinate devices. It establishes the connection with the CSP subordinate sub-systems observing devices. On success, the main sub-systems attributes are subscribed for events.

It is invoked when the adminMode of the device is OFFLINE or MAINTENANCE.

**stop_communicating**() → None

Method invoked when the CSP Subarray TANGO Device is set OFFLINE.

In this case the CspSubarrayComponentManager stop monitoring the subordinate devices.

The admin mode is forwarded to all the online subordinate components. If the request fails, only the component offline are disconnected.

**on**(*command_id: str*) → Tuple[ska_tango_base.commands.ResultCode, str]

Enable the CSP Subarray to perform observing tasks. The command is forwarded to all the CSP Subsystems observing devices (subarrays or beams) that are currently online.

**Returns**
a tuple with *ResultCode* and an informative message.

**off**(*command_id: str*) → Tuple[ska_tango_base.commands.ResultCode, str]

Disable the CSP Subarray to perform observing tasks. The command is forwarded to all the CSP Subsystems observing devices (subarrays or beams) that are currently online.

The command can be invoked from *any* observing state with the result that any ongoing process, is aborted and all the assigned resources released.

**Returns**
a tuple with *ResultCode* and an informative message.

**configure**(*argin: dict*) → Tuple[ska_tango_base.commands.ResultCode, str]

Configure the CSP Subarray to perform observing tasks. The command is forwarded to all the CSP Subsystems observing devices (subarrays or beams) that are currently online.

**Parameters**
**argin** – The CSP Subarray configuration data

**Returns**
a tuple with *ResultCode* and an informative message.

**gotoidle**() → Tuple[ska_tango_base.commands.ResultCode, str]

Force the CSP Subarray to IDLE to allow changes in the resources assigned to the subarray. The command is forwarded to al the online CSP sub-systems.

**Returns**
a tuple with *ResultCode* and an informative message.

**abort**() → Tuple[ska_tango_base.commands.ResultCode, str]

Abort any running stoppable CSP subarray task. The command is forwarded to all the CSP Sub-systems observing devices (subarrays or beams) that are currently online.

**Returns**
a tuple with *ResultCode* and an informative message.

**obsreset**() → Tuple[ska_tango_base.commands.ResultCode, str]

Reset a FAULT or ABORTED condition of the CSP Subarray. The command is forwarded to all the CSP Sub-systems observing devices (subarrays or beams) that are currently online and in FAULT or ABORTED observing state.

**Returns**
a tuple with *ResultCode* and an informative message.

**restart**() → Tuple[ska_tango_base.commands.ResultCode, str]

Restart a CSP Subarray in FAULT or ABORTED observing state.

The command is forwarded to all the CSP Sub-systems observing devices (subarrays or beams) that are currently online.

**Returns**
a tuple with *ResultCode* and an informative message.

**assignresources**(*argin: dict*, *command_id: str*) → Tuple[ska_tango_base.commands.ResultCode, str]

Load the configuration from file specifying the resources for subsystems. Forward the command to all the subsystems.

**Parameters**
**argin** – the information with the resources to allocate to the CSP Subarray

**Returns**
a tuple with *ResultCode* and an informative message.

**releaseresources**(*argin: dict*, *command_id: str*) → Tuple[ska_tango_base.commands.ResultCode, str]

Forward the command to release assigned resounces to all the subsystems.

**Parameters**
**argin** – the information with the resources to remove from the CSP Subarray

**Returns**
a tuple with *ResultCode* and an informative message.

releaseallresources(*command_id: str*) → Tuple[ska_tango_base.commands.ResultCode, str]

Forward the command to release all assigned resources to all the subsystems.

**Returns**
a tuple with *ResultCode* and an informative message.

off_completed()

Method invoked on task completion. If the CSP Subarray has some timing beams assigned, these are disconnected, the attributes unsubscribed and the list updated.

scan(*argin: dict*) → Tuple[ska_tango_base.commands.ResultCode, str]

Start a Scan. The command is forwarded to all the CSP Sub-systems observing devices (subarrays or beams) that are currently online.

**Parameters**
**argin** – the scan information.

**Returns**
a tuple with *ResultCode* and an informative message.

endscan() → Tuple[ska_tango_base.commands.ResultCode, str]

Stop gracefully the execution of a scan.

The command is forwarded to all the CSP Sub-systems observing devices (subarrays or beams) that in scanning.

**Returns**
a tuple with *ResultCode* and an informative message.

### Event Manager

class ska_csp_lmc_common.manager.**EventManager**(*health_model: HealthStateModel*, *op_state_model: OpStateModel*, *obs_state_model: Optional[ObsStateModel] = None*, *max_queue_size: Optional[int] = 100*, *logger: Optional[Logger] = None*)

Bases: BaseEventManager

Aggregator class to manage the event subscriptions.

__init__(*health_model: HealthStateModel*, *op_state_model: OpStateModel*, *obs_state_model: Optional[ObsStateModel] = None*, *max_queue_size: Optional[int] = 100*, *logger: Optional[Logger] = None*)

Initialize the EventManager for the current device.

Handles the subscription and un-subscription of attributes on the CSP sub-system components. It also performs the aggregation of tha main attributes (State, healthState and obsState) of the CSP subordinate sub-systems to evaluate the state of the CSP as a whole.

**Parameters**
**logger** – The device or python logger if default is None.

_process_event(*evt: CspEvent*) → None

Handle the event fetched from the queue.

**Parameters**
**evt** – an object with the event data

**_evaluate_csp_state**(*evt: CspEvent*) → None

Evaluate the CSP Device State as aggregation of the state of the subordinate components.

It also re-evaluates the CSP Device healthState that can be affected by the change in the state of one or more CSP sub-system components.

This method relies on the device OpStateModel to perform the aggregation of the CSP sub-systems operational states.

**Parameters**
**evt** – an object containing the event data.

**_evaluate_csp_obsstate**(*evt: CspEvent*) → None

evaluate the CSP Device observing state as aggregation of the observing state of the CSP subordinate subsystems.

This method relies on the device ObsStateModel to perform the aggregation of the CSP sub-systems observing states.

**Parameters**
**evt** – the CspEvent received from a subordinate component

**_evaluate_csp_healthstate**(*evt: CspEvent*) → None

Evaluate the CSP Device health state as aggregation of the health state of the CSP subordinate sub-systems.

This method relies on the device HealthStateModel to perform the aggregation of the CSP sub-systems health values.

**Parameters**
**evt** – the CspEvent received from a subordinate component

### Component Manager Configurator

**class** ska_csp_lmc_common.manager.manager_configuration.**ComponentManagerConfiguration**(*dev_name: str*, *logger: Optional[Logger] = None*)

Bases: object

Class to store the device properties of the controlling TANGO Device to pass to the ComponentManager.

**__init__**(*dev_name: str*, *logger: Optional[Logger] = None*) → None

Initialize the class with the name of the device. The device name is needed to retrieve the device properties from the TANGO DB.

**Parameters**

- **dev_name** – the device name for which the device properties list is retrieved from the TANGO DB
- **logger** – a logger for this instance

**get_device_properties**() → Dict[str, str]

Retrieve the list of the Tango properties of the device registered within the TANGO DB.

Format the information as a dictionary where each entry is the property name and the value is the property value (as a string).

**Returns**

A dictionary with the property name and the associated value.

**add_attributes**() → None

Add the device properties as attribute of the class.

### 3.2.2 CSP Sub-system Component

#### Component

**class** ska_csp_lmc_common.component.**Component**(*fqdn: str*, *name: str*, *weight: int = 0*, *logger=None*)

Bases: object

Interface class to a sub-system device.

**__init__**(*fqdn: str*, *name: str*, *weight: int = 0*, *logger=None*) → None

Initialize the component instance.

**Parameters**

- **fqdn** – the sub-system FQDN
- **name** – the component name (for ex. 'cbf-ctrl', 'pss-ctrl', 'pst-beam-1', etc)
- **name** – string
- **weight** – the sub-system 'weight'. CBF sub-system has an higher impact on the CSP.LMC functionalities.
- **logger** – a logger for this instance

**__hash__**()

Define the __hash__() method for the Component class to use this object as a key in a python dictionary.

**__eq__**(*other: Component*)

Define the __eq__() method for the Component class to use this object as a key in a python dictionary.

**__key**()

Define the __key() method for the Component class to use this object as a key in a python dictionary.

**property event_id: List[int]**

Return the list of registered events.

**Returns**

A list with the ID of the events subscribed on the component.

**property event_attrs: List[str]**

Return the list of attributes subscribed for events.

**Returns**

A list with the attributes subscribed on the component.

**property fqdn**

Return the FQDN of the sub-system associated to the current component.

**property proxy**

Return the DeviceProxy with the CSP sub-system TANGO device if this is reachable, otherwise None.

**property state**

Return the sub-system state.

**Returns**
the sub-system State if updated via events or via direct read, UNKNOWN on failure

**property health_state**

Return the sub-system health_state.

**Returns**
the sub-system healthState if updated via events or via direct read, UNKNOWN on failure

**property admin_mode**

Return the sub-system health_state.

**Returns**
the sub-system adminmode if updated via events or via direct read, UNKNOWN on failure

**_get_attribute**(*attr_name: str*) → Any

Return the value of the required attribute. If the attribute is not initialized, its value is retrieved via direct read on the sub- system.name.

**Parameters**
**attr_name** – the name of the attribute

**Returns**
the attribute value on success, None otherwise

**_update_component_info**(*recv_evt: tango.EventData*, **new_evt: CspEvent*) → bool

Method to update the sub-system component manager internal status when an event generated by the sub-system TANGO device is caught.

**Parameters**

- **recv_evt** – the event generated by the CSP sub-system TANGO device
- **new_evt** – the eve/nt forwarded back to the CSP TANGO device.

**Returns**
True to forward the event value back to the CSP device.

**_handle_event_errors**(*recv_event*, *fwd_event*)

Method to handle the error conditions on received events. Events with errors are not always propagated back to the main CSP device.

**Parameters**
**recv_event** – the received event

**Returns**
True if the received event is forwarded back to the CSP device, otherwise False

**_push_event**(*recv_event: tango.EventData*) → None

Callback function invoked when an event is received. The method checks for errors: when a *loss of connection* is detected, the value of the attribute is set to UNKNOWN, if the attribute support this value, otherwise to None (with quality factor set to INVALID). In the first case the attribute is updated inside the component class, too. After all checks, the method invokes the callback register at supscription, if any, passing as argument an instance of the CspEvent class with the new value.

**Parameters**
**recv_event** – The received event data class

**Returns**
None

**set_component_disconnected**()

This method is called when the CSP TANGO Device adminMode is set to OFFLINE.

In this case the CSP Device componentManager does no longer monitor the component and its information are reported as unknown. The component admin mode is left unchanged.

**set_component_unknown**(*admin_mode_value: ska_tango_base.control_model.AdminMode*) → None

This method is called when the component experiences a loss of connection. In this context, this method sets the State and healthState attribute to UNKNOWN. For the other attributes, the value is set to the default value None and quality_factor to ATTR_INVALID.

**Parameters**
**args** – an instance of the CspEvent class with the new values.

**Returns**
None

**set_component_offline**(*admin_mode_value: ska_tango_base.control_model.AdminMode*) → None

This method is called when the received event is related to a device not registered into the DB or its admin mode is OFFLINE/NOT-FITTED In this context, this method sets the State or healthState attribute to UNKNOWN. For the other attributes, the value is set to the default value None and quality_factor to ATTR_INVALID.

**Parameters**
**admin_mode_value** – the value of the CSP sub-system device adminMode.

**Returns**
None

**connect**() → Connector | None

Establish the connection with a sub-system device. Connection retries happen with a interval configured via the device property PingConnectionTime. If the subordinate device is not registered into the TANGO DB, the CSP Controller/Subarray device tries connection up to 3 times before throwing an exception and considering the sub-system not on-line (available). This approach is related to the deployment procedure: each sub-system configures the TANGO DB independently, through a configurator process. It may happen that the CSP Controller/Subarray is already running while the configurator of one or more sub-systems is still writing the TANGO DB. In this case, the CSP would not be able to detect the sub-system because the DB is fully configured. Retry operations provide more time to wait for the end of the TANGO DB configuration.

**Returns**
The Connector on success, otherwise None

**Raise**
a DevFailed exception on connection failure.

**disconnect**() → None

Invalidate the connection with the CSP Sub-subsystem and report the main SKA SCM attributes accordingly to the expected values.

**read_attr**(*attribute: str*) → Any

Return the value of the requested attribute.

**Parameters**
**attribute** – the attribute name

**Returns**
the attribute value, if the attribute does exist on the sub-system device

**Raise**
a ValueError exception if the attribute does not exist or read failure.

**write_attr**(*attribute: str*, *value: Any*) → None

Set the value of the requested attribute.

**Parameters**

- **attribute** – the attribute name
- **value** – the value to set

**Raise**
a ValueError exception if the attribute does not exist

**force_attribute_update**(*attr_name: str*) → None

Update the attribute via a direct read. it also invokes the _push_event method to the EventManager internal attribute value.

**Parameters**
**attr_name** – the name of the attribute forced to read.

**run**(*command_name: str*, *async_flag: bool = True*, *argument: Optional[Any] = None*, *callback: Optional[Callable] = None*) → None

Execute a command on the target device.

**Parameters**

- **command_name** – the command name
- **async_flag** – set the execution model (async/sync)
- **argument** – the command argument, if any
- **callback** – callable called when the command ends on the target device, if any

**read_timeout**(*command_name: str*) → int

Read the timeout configured for a command.

**Parameters**
**command_name** – the command name

**Returns**
the timeout configured (in secs) or 0 on failure

**subscribe_attribute**(*attr_name: str*, *event_type: tango.EventType*, *evt_mgr_callback: Optional[Callable] = None*) → bool

Subscribe to any event.

**Parameters**

- **attr_name** – the attribute name
- **event_type** – the event type (CHANGE_EVENT, PERIODIC, etc..)
- **evt_mgr_callback** – the EventManager method called when the event is received.

**Returns**
True on subscription success, otherwise False.

**unsubscribe_attribute**(*attr_list: Optional[List[str]] = None*) → None

Unsubscribe the event on the specified attributes. If the event_id dictionary is empty, the event_callback is un-registered.

**Parameters**
**attr_list** – the list of attributes to un-subscribe event on. If the list is empty, all the subscribed events on the sub-system are un-subscribed.

**Returns**
None (?) <=== CHECK

**on**(*callback: Optional[Callable] = None*) → None

Define the behavior of the On command for a component. Override this method as required. This method has been specialized to work with CSP sub-systems Controllers devices.

**Parameters**
**callback** – callable object invoked when command completes on the target TANGO Device.

**standby**(*callback: Optional[Callable] = None*) → None

Define the behavior of the Standby command for a component. Override this method as required. This method has been specialized to work with CSP sub-systems Controllers devices.

**Parameters**
**callback** – callable object invoked when command completes on the target TANGO Device.

**off**(*callback: Optional[Callable] = None*) → None

Define the behavior of the Off command for a component. Override this method as required. This method has been specialized to work with CSP sub-systems Controllers devices.

**Parameters**
**callback** – callable object invoked when command completes on the target TANGO Device.

### Observing Component

### CBF Controller Component

**class** ska_csp_lmc_common.controller.cbf_controller.**CbfControllerComponent**(*fqdn*, *logger=None*)

Bases: Component

### PSS Controller Component

**class** ska_csp_lmc_common.controller.pss_controller.**PssControllerComponent**(*fqdn: str*, *logger: Optional[Logger] = None*)

Bases: Component

### PST Controller Component

### CBF Subarray Component

class ska_csp_lmc_common.subarray.cbf_subarray.**CbfSubarrayComponent**(*fqdn*, *logger=None*)

Bases: ObservingComponent

**assignresources**(*resources_dict: Dict*, *callback: Optional[Callable] = None*) → None

Run the command assignresources on the component.

**Raise**
a ValueError exception if the component is not in the proper state to run the command

**releaseresources**(*resources_dict: Dict*, *callback: Optional[Callable] = None*) → None

Run the command releaseresources on the component.

Method to specialize into the specific component, if needed.

**Raise**
a ValueError exception if the component is not in the proper state to run the command

**releaseallresources**(*callback: Optional[Callable] = None*)

Run the command releaseallresources on the component.

Method to specialize into the specific component, if needed.

**Raise**
a ValueError exception if the component is not in the proper state to run the command

**scan**(*scan_data: Dict*, *callback: Callable*) → None

Invoke the *Scan* command on the Mid CBF subarray.

**Parameters**

- **scan_data** – The dictionary with scan data
- **callback** – Method invoked when the commands end on the target device

**update_pst_json_configuration**(*original_dict*, *updated_info*)

**configure**(*resources_dict: Dict*, *callback: Optional[Callable] = None*) → None

Specialization of the method for the CBF Subarray

### PSS Subarray Component

class ska_csp_lmc_common.subarray.pss_subarray.**PssSubarrayComponent**(*fqdn: str*, *logger: Optional[Logger] = None*)

Bases: ObservingComponent

**assignresources**(*resources_dict: Dict*, *callback: Optional[Callable] = None*) → None

Run the command assignresources on the component.

**Raise**
a ValueError exception if the component is not in the proper state to run the command

**releaseresources**(*resources_dict: Dict*, *callback: Optional[Callable] = None*) → None

Run the command releaseresources on the component.

Method to specialize into the specific component, if needed.

**Raise**
a ValueError exception if the component is not in the proper state to run the command

**releaseallresources**(*callback: Optional[Callable] = None*) → None

Run the command releaseallresources on the component.

Method to specialize into the specific component, if needed.

**Raise**
a ValueError exception if the component is not in the proper state to run the command

**scan**(*argument: Dict*, *callback: Optional[Callable] = None*) → None

Define the behavior of the Scan command for a component. Override this method as required.

**Parameters**

- **argument** – the input argument
- **callback** – callable object invoked when command completes on the target TANGO Device.

### PST Beam Component

**class** ska_csp_lmc_common.subarray.pst_beam.**PstBeamComponent**(*fqdn: str*, *logger: Optional[Logger] = None*)

Bases: ObservingComponent

Adaptor class for a PstBeam device.

**property subarray_id**

Return the subarray membership.

**releaseresources**(*resources_dict: Dict*, *callback: Optional[Callable] = None*) → None

Specialization of the method for a PstBeam

**property channel_block_configuration**

Return the channel block configuration

**releaseresources_succeeded**()

ReleaseResources succeeded method for a PstBeam

**releaseallresources**(*callback: Optional[Callable] = None*) → None

Specialization of the method for a PstBeam

**releaseallresources_succeeded**()

ReleaseAllResources succeeded method for a PstBeam

**assignresources**(*resources_dict: Dict*, *callback: Optional[Callable] = None*) → None

Specialization of the method for a PstBeam.

The connection with the PstBeam Tango device is established.

**assignresources_succeeded**()

AssignResources succeeded method for a PstBeam

**configure**(*resources_dict: Dict*, *callback: Optional[Callable] = None*) → None

Specialization of the method for a PstBeam.

**scan**(*resources_dict: Dict*, *callback: Optional[Callable] = None*) → None

Specialization of the method for a PstBeam.

### 3.2.3 Command subpackage

#### Component Command

class ska_csp_lmc_common.commands.component_command.**ComponentCommand**(*name: str*, *component: Component*, *resources: Optional[Any] = None*, *logger: Optional[Logger] = None*)

Bases: BaseComponentCommand

Abstract class to model a sub-system (or *component*) command.

A component command operates on a CSP subordinate sub-system.

**__init__**(*name: str*, *component: Component*, *resources: Optional[Any] = None*, *logger: Optional[Logger] = None*) → None

Class *init* method.

**Parameters**

- **name** – the command name
- **component** (Component) – the subsystem component on which the command acts
- **resources** – the command input argument, if any (default None)
- **logger** – the device logger

**property component: Component**

Return the sub-system receiver of the command.

**Returns**
the sub-system component the command acts on

**run**() → None

Method to execute the command on the receiver component.

**failure_detected**()

Return whether a failure was detected during command execution.

Failure conditions are: - Timeout expired - Failure in command execution on a sub-system - Sub-system reports FAULT obsState.

**command_ended**()

Helper function called when the command ends on the sub-system component. It evaluates the command execution time and invokes the observer.

*notify* method.

**_command_monitor**() → None

Thread target function.

Issue the command on the target sub-system component and monitor the command status, waiting for the end of the command. The process is regulated via a timeout. The command completion (either with success or failure) is notified to the main command observer (instantiate into the ComponentManager) invoking the *notify* method on it. On exception, the failure_raised flag is set.

**_run**(*argument: Optional[Any] = None*, *callback: Optional[Callable] = None*) → None

If not specialized simply calls the run method on the corresponding component.

### Base Component Commands

class ska_csp_lmc_common.commands.base_commands.**ComponentInit**(*component: Component*, *resources: Optional[Any] = None*, *logger: Optional[Logger] = None*)

Class for handling the Subarray and Controller initialization and re- initialization.

**set_timeout**(*timeout_value: int*) → None

Configure the maximum timeout for the connection.timeout.

**Parameters**
**timeout_value** – the maximum time to wait for the connection.

**_command_monitor**() → None

Base method overridden to handle the connection with a CSP sub- system TANGO devices.

**succeeded**()

Does nothing but needed because inherit abstract class.

class ska_csp_lmc_common.commands.base_commands.**ComponentDisconnect**(*component: Component*, *resources: Optional[Any] = None*, *logger: Optional[Logger] = None*)

Class for handling the Subarray and Controller initialization and re- initialization.

**_run**(*argument: Optional[Any] = None*, *callback: Optional[Callable] = None*) → None

If not specialized simply calls the run method on the corresponding component.

**succeeded**()

Does nothing but needed because inherit abstract class.

class ska_csp_lmc_common.commands.base_commands.**ComponentOn**(*receiver: Component*, *logger: Optional[Logger] = None*)

Class specialized to handle the On of a sub-system.

**_run**(*argument: Optional[List] = None*, *callback: Optional[Callable] = None*) → None

If not specialized simply calls the run method on the corresponding component.

**succeeded**() → bool

Return whether the command executed on a CSP sub-system completes successfully.

class ska_csp_lmc_common.commands.base_commands.**ComponentOff**(*receiver: Component*, *logger: Optional[Logger] = None*)

Class specialized to handle the Off of a sub-system.

**_run**(*argument: Optional[List] = None*, *callback: Optional[Callable] = None*) → None

If not specialized simply calls the run method on the corresponding component.

**succeeded**() → bool

Return whether the command executed on a CSP sub-system completes successfully.

class ska_csp_lmc_common.commands.base_commands.**ComponentStandby**(*receiver: Component*, *logger: Optional[Logger] = None*)

Class specialized to handle the Standby of a sub-system.

**_run**(*argument: Optional[Any] = None*, *callback: Optional[Callable] = None*) → None

If not specialized simply calls the run method on the corresponding component.

**succeeded**()

Return whether the command executed on a CSP sub-system completes successfully.

class ska_csp_lmc_common.commands.base_commands.**ComponentReset**(*receiver: Component*, *logger: Optional[Logger] = None*)

Class specialized to handle the Reset of a sub-system.

**succeeded**() → bool

Return whether the command executed on a CSP sub-system completes successfully.

class ska_csp_lmc_common.commands.base_commands.**ComponentNop**(*receiver: Component*, *logger: Optional[Logger] = None*)

Class for Not Operative Command (NOP).

**property expected_condition**

Return the expected condition

**endscan_request_pending**()

Whether the endscan event is set

**_run_nop_command**()

Whether the NOP command can be executed

When a NOP command is part of an interruptable command (Configure, Scan, ObsReset) the execution of its run method is skipped when:

- no event is set
- **an event is set but the command the NOP command is related to** another event

**run**() → None

Execute a wait loop.

This command, when not skipped, waits for a specific condition.

**succeeded**() → bool

Return whether the required conditions are satisfied.

If no conditions are specified, the command waits for the configured timeout.

### Observing Component Commands

This module includes the ComponentCommand specialized classes.

### AssignResources

class ska_csp_lmc_common.commands.observing_commands.**ComponentAssign**(*receiver: ObservingComponent*, *resources: str*, *logger: Optional[Logger] = None*)

Bases: ComponentCommand

Class to handle the *AssignResources* command on a CSP sub-system component.

**_run**(*argument: Dict*, *callback: Optional[Callable] = None*) → None

The method is specialized to invoke the *AssignResources* command on the CSP sub-system component:

self._component.assignresources(argument, callback=callback)

**succeeded**() → bool

Return whether the command executed on a CSP sub-system completes successfully.

The *AssignResources* command on a sub-system ends with success when the following condition is satisfied:

self._component.obs_state == ObsState.IDLE

### ReleaseResources

**class** ska_csp_lmc_common.commands.observing_commands.**ComponentRelease**(*component: ObservingComponent*, *resources: Optional[Any] = None*, *logger: Optional[Logger] = None*)

Bases: ComponentCommand

Class to handle *ReleaseResources* command on a CSP sub-system component.

**_run**(*argument: dict*, *callback: Callable*) → None

The method is specialized to invoke the *ReleaseResources* command on the CSP sub-system component:

self._component.releaseresources(argument, callback=callback)

**succeeded**() → bool

Return whether the command executed on a CSP sub-system completes successfully.

The *ReleaseResources* command on a sub-system ends with success when the following condition is satisfied:

self._component.obs_state == ObsState.EMPTY or self._component.obs_state == ObsState.IDLE

### ReleaseAllResources

**class** ska_csp_lmc_common.commands.observing_commands.**ComponentReleaseAll**(*component: ObservingComponent*, *logger: Optional[Logger] = None*)

Bases: ComponentCommand

Class to handle Subarray *ReleaseAllResources* command.

**_run**(*argument: Optional[Any] = None*, *callback: Optional[Callable] = None*) → None

The method is specialized to invoke the *ReleaseAllResources* command on the CSP sub-system component:

self._component.releaseallresources(callback=callback)

**succeeded**()

Return whether the command executed on a CSP sub-system completes successfully.

The *ReleaseAllResources* command on a sub-system ends with success when the following condition is satisfied:

component.obs_state == ObsState.EMPTY

### Configure

**class** ska_csp_lmc_common.commands.observing_commands.**ComponentConfigure**(*component: ObservingComponent*, *resources*, *logger: Optional[Logger] = None*)

Bases: ComponentCommand

Class to handle *Configure* command on a CSP sub-system component.

**_run**(*argument: Dict*, *callback: Optional[Callable] = None*) → None

Invoke the command on the CSP sub-system.

**succeeded**() → bool

Return whether the command executed on a CSP sub-system completes successfully. The *Configure* command on a sub-system ends with success when the following condition is satisfied:

self._component.obs_state == ObsState.READY

**failure_detected**()

Return whether a failure was detected during command execution.

Failure conditions are: - Timeout expired - Failure in command execution on a sub-system - Sub-system reports FAULT obsState.

### Scan

**class** ska_csp_lmc_common.commands.observing_commands.**ComponentScan**(*component: ObservingComponent*, *resources: Any*, *logger: Optional[Logger] = None*)

Bases: ComponentCommand

Class to handle Scan command on a CSP sub-system component.

**_run**(*argument: Any*, *callback: Optional[Callable] = None*) → None

The method is specialized to invoke the *Scan* command on the CSP sub-system component.

**_command_monitor**()

Base method overridden.

**succeeded**() → None

This command does not specialize the *succeeded* method.

**terminated**() → bool

Whether the scan terminated conditions occurred.

### EndScan

class ska_csp_lmc_common.commands.observing_commands.**ComponentEndScan**(*component: ObservingComponent*, *logger: Optional[Logger] = None*)

Bases: ComponentCommand

Class to handle the EndScan command on a sub-system component.

**_command_monitor**() → None

Thread target function for abort command.

Specialization of the base _command_monitor() method.

**succeeded**()

Return whether the command executed on a CSP sub-system completes successfully.

The *EndScan* command on a sub-system ends with success when the following condition is satisfied:

self._component.obs_state == ObsState.READY

**failure_detected**()

Return whether a failure was detected during command execution.

Failure conditions are: - Timeout expired - Failure in command execution on a sub-system - Sub-system reports FAULT obsState.

### Abort

class ska_csp_lmc_common.commands.observing_commands.**ComponentAbort**(*component: ObservingComponent*, *logger: Optional[Logger] = None*)

Bases: ComponentCommand

Class to handle Abort command on a sub-system component.

**succeeded**()

Return whether the command executed on a CSP sub-system completes successfully.

The *Abort* command on a sub-system ends with success when the following condition is satisfied:

self._component.obs_state == ObsState.ABORTED

### ObsReset

class ska_csp_lmc_common.commands.observing_commands.**ComponentObsReset**(*component: ObservingComponent*, *logger: Optional[Logger] = None*)

Bases: ComponentCommand

Class to handle Subarray ObsReset command.

**succeeded**() → bool

Return whether the command executed on a CSP sub-system completes successfully.

The *Abort* command on a sub-system ends with success when the following condition is satisfied:

self._component.obs_state == ObsState.IDLE

### Restart

**class** ska_csp_lmc_common.commands.observing_commands.**ComponentRestart**(*component: ObservingComponent*, *logger: Optional[Logger] = None*)

Bases: ComponentCommand

Class to handle Subarray Restart command.

**succeeded**() → bool

Return whether the command executed on a CSP sub-system completes successfully.

The *Restart* command on a sub-system ends with success when the following condition is satisfied:

self._component.obs_state == ObsState.EMPTY

### Macro Component Commands

**class** ska_csp_lmc_common.commands.macro_command.**MacroComponentCommand**(*name: str*, *command_factory: CommandFactory*, *notify_callback: Callable*, *macro_items: List[MacroCommandItem]*, *logger: Optional[Logger] = None*, *abort_event: Optional[Event] = None*)

Bases: BaseComponentCommand

Class modeling a Macro command.

A Macro-command is a set of sub-system or *component* commands grouped together as a single command and executed in sequence, one after the other, to accomplish a CSP task. Sub-system commands can have different sub-system targets.

**__init__**(*name: str*, *command_factory: CommandFactory*, *notify_callback: Callable*, *macro_items: List[MacroCommandItem]*, *logger: Optional[Logger] = None*, *abort_event: Optional[Event] = None*)

**Parameters**

- **name** – the name of the macro command.
- **macro_items** – a list of MacroCommandItem entries
- **command_factory** – the factory class to create the CSP Component commands classes.

- **logger** – the device logger target

**property command_in_execution**

Return the command currently in execution.

**property is_running: bool**

Whether the macro-command is running.

The macro command is running if at least one of its components command is in running state

TO REMOVE when move to BC13

**property failure_raised: bool**

Whether the macro-command failure flag is set.

The macro command failure flag is set when at least one of its components command is failed.

TO REMOVE when move to BC13

**property timeout_expired: bool**

Whether the macro-command timeout expired flag is set.

The macro command timeout flag is set when at least one of its components command elapsed its timeout.

**property aborted: bool**

Flag to report whether an abort request has been process by the CSP sub-system.

**Returns**
whether an abort request has been processed by the sub-system.

**add**(*command: ComponentCommand*)

Add a sub-task command

**tag**()

Tag the command of the macro command Each component command is tagged with two numbers: - the order in the list of execution - the total number of commands belonging to the macro command

**command_ended**()

Invoked on command completion.

**status_ok**()

Invoked on command completion.

**_command_monitor**()

Method to run and monitor the execution of a macro command.

Component commands are executed one after the other. Each component command notifies its end to the command observer. If one command fails, the other commands are skipped.

TODO: This behavior does not apply to all the type of commands: need to configure this functionality.

**run**() → None

Method to execute the command on the receiver component.

### 3.2.4 CSP State Models

#### Operational State Model

class ska_csp_lmc_common.model.**OpStateModel**(*op_state_init: tango.DevState*, *op_state_changed_callback: Callable[[tango.DevState], None]*, *logger: Optional[Logger] = None*)

Bases: object

A simple operational state model.

- DevState.DISABLE – when communication with the component is not established.
- DevState.FAULT – when the component has faulted

**__init__**(*op_state_init: tango.DevState*, *op_state_changed_callback: Callable[[tango.DevState], None]*, *logger: Optional[Logger] = None*) → None

Initialise a new instance.

**Parameters**

- **op_state_init** – the initial state of the component under control.
- **op_state_changed_callback** – callback to be called whenever there is a change to evaluated operational state.
- **logger** – a logger for this instance

**property faulty**

Return whether the componend is experiencing a faulty condition or not.

**Returns**
whether the component is in fault.

**property disabled: bool**

Return whether the component is disabled or not.

**Returns**
whether the component is disabled.

**property op_state: tango.DevState**

Return the component operational state.

**Returns**
the operational state.

**update_op_state**() → None

Update operational state.

This method calls the :py:meth:`evaluate_op_state` method to figure out what the new operational state should be, and then updates the `state` attribute, calling the callback if required.

**evaluate_op_state**() → tango.DevState

Re-evaluate the operational state.

This method contains the logic for evaluating the state.

This method should be extended by subclasses in order to define how state is evaluated by their particular device.

If the CSP device opState is in FAULT for an internal error (i.e not depending from the opStates of the CSP sub-systems) this state has to be maintained. The only way to exit from this state is to Reset/Reinit the CSP device. In this case the faulty flag is reset to False.

**Returns**
the new state state.

**component_fault**(*faulty: bool*) → None

Handle a component experiencing or recovering from a fault.

This method is called when the component goes into or out of FAULT state.

**Parameters**
**faulty** – whether the component has faulted or not

**is_disabled**(*disabled: bool*) → None

Handle disabling the monitoring functionalities of a TANGO device.

This method is called when the communication between the TANGO device and the component under controller is disabled/enabled via the setting of the adimnistrative mode.

**Parameters**
**disabled** – whether the communication between the component and the controlling TANGO device is disabled.

### Health State Model

**class** ska_csp_lmc_common.model.**HealthStateModel**(*init_state: ska_tango_base.control_model.HealthState, health_changed_callback: Callable[[ska_tango_base.control_model.HealthState], None], logger: Optional[Logger] = None*)

Bases: object

A simple health model the supports.

- HealthState.OK – when the component is fully operative.
- HealthState.DEGRADED – when the component is partially operative.
- HealthState.UNKNOWN – when communication with the component is not established.
- HealthState.FAILED – when the component has faulted

**__init__**(*init_state: ska_tango_base.control_model.HealthState, health_changed_callback: Callable[[ska_tango_base.control_model.HealthState], None], logger: Optional[Logger] = None*) → None

Initialise a new instance.

**Parameters**

- **init_state** – The health state of the component under control at initialization.
- **health_changed_callback** – callback to be called whenever there is a change to this this health model's evaluated health state.
- **logger** (*an instance of :py:class`logging.Logger`, or an object that implements the same interface*) – a logger for this instance

**property faulty**

Return whether the componend is experiencing a faulty condition or not.

**Returns**
whether the component is in fault.

**property disabled**

Return whether the component is disabled or not.

**Returns**
whether the component is disabled

**property health_state: ska_tango_base.control_model.HealthState**

Return the health state of the component under control.

**Returns**
the health state.

**update_health**() → None

Update health state.

This method calls the :py:meth:`evaluate_health` method to figure out what the new health state should be, and then updates the `health_state` attribute, calling the callback if required.

**evaluate_health**() → ska_tango_base.control_model.HealthState

Re-evaluate the health state.

This method contains the logic for evaluating the health.

This method should be extended by subclasses in order to define how health is evaluated by their particular device.

**Returns**
the new health state.

**component_fault**(*fault: bool*) → None

Handle a component experiencing or recovering from a fault.

This is a callback hook that is called when the component goes into or out of FAULT state.

**Parameters**
**fault** – whether the component has faulted or not

**is_disabled**(*disabled: bool*) → None

Handle change in communication with the component.

**Parameters**
**disabled** – whether communications with the component is established.

### Observing State Model

**class** ska_csp_lmc_common.model.**ObsStateModel**(*obs_state_init: ska_tango_base.control_model.ObsState, obs_state_changed_callback: Callable[[ska_tango_base.control_model.ObsState], None], logger: Optional[Logger] = None*)

Bases: `object`

A simple observing state model for observing devices.

__init__(*obs_state_init: ska_tango_base.control_model.ObsState*, *obs_state_changed_callback: Callable[[ska_tango_base.control_model.ObsState], None]*, *logger: Optional[Logger] = None*) → None

Initialise a new instance.

**Parameters**

- **obs_state_int** – the observing state of the component under control at initialization.
- **obs_state_changed_callback** – callback to be called whenever the observing state of the component under control (as evaluated by this model) changes.
- **logger** – a logger for this instance

property faulty

Return whether the component is experiencing a faulty condition or not.

**Returns**
whether the component is in fault.

property action_driven

Return whether the updating of the component observing State is driven by actions or events (default).

**Returns**
whether the component obseving state update is driven by actions or events.

property obs_state: ska_tango_base.control_model.ObsState

The observing state of the component under control of the CSP Subarray TANGO Device.

**Getter**
the current observing state

**Setter**
set the component observing state to the updated value and the device callback is invoked, if defined.

_update_obs_state(*obs_state*) → None

Helper method to handle the update of the observing state of a component and the device that controls it.

# pylint: disable-next=fixme *TODO*: lock the TANGO Device AutoTangoMonitor otherwise there race conditions can happen.

update_obs_state() → None

Update the component observing state.

This method calls the :py:meth:`evaluate_obs_state` method to figure out what the new obsstate state should be, and then updates the `obs_state` attribute, calling the callback if required.

evaluate_obs_state() → ska_tango_base.control_model.ObsState

Re-evaluate the component observing state.

This method contains the basic logic for evaluating the observing state.

This method should be extended by subclasses in order to define how observing state is evaluated by their particular device.

**Returns**
the new observing state.

component_fault(*faulty: bool*) → None

Handle a component experiencing or recovering from a fault.

This method is called when the component goes into or out of FAULT state.

**Parameters**
**faulty** – whether the component has faulted or not.

**component_disabled**(*disabled: bool*) → None

Handle the monitoring functionalities of a TANGO Device.

This method is called when the communication between the TANGO device and the component under controller is disabled/enabled via the setting of the adimnistrative mode.

**Parameters**
**disabled** – whether the communication between the component and the controlling device is disabled.

**component_action_driven**(*action_driven: bool*) → None

Whether the updating of the component observing state is driven by actions or events (default).

**Parameters**
**action_driven** – configure the model behavior to update the observing state.

### Controller Operational State Model

**class** ska_csp_lmc_common.controller.controller_op_state.**ControllerOpStateModel**(*op_state_init: tango.DevState*, *op_state_changed_callback: Callable[[tango.DevState], None]*, *logger: Optional[Logger] = None*)

Bases: `OpStateModel`

A simple operational state model that supports.

- DevState.ON – when the component is powered on.
- DevState.OFF – when the component is powered off.
- DevState.STANDBY – when the component is low-power mode.
- DevState.ON – when the component is powered on.
- DevState.DISABLE – when the component is in OFFLINE administrative mode.
- DevState.UNKNOWN – when communication with the component is not established.
- DevState.FAILED – when the component has faulted

**property op_state:** *tango.DevState*

Return the operational state.

**Returns**
the operational state state.

**Return type**
DevState

**update_op_state**() → None

Update the operational state.

This method calls the :py:meth:`evaluate_op_state` method to figure out what the new operational state should be, and then updates the `op_state` attribute, calling the TANGO device callback if required.

**evaluate_op_state**() → tango.DevState

Compute overall operational state of the CSP controller device based on the fault and communication status of the controller overall, together with the aggregation of the operational states of the subordinate sub-systems components.

**Returns**
an overall operational state of the controller.

**Return type**
DevState

**component_op_state_changed**(*component: Component*, *op_state: DevState | None*) → None

Handle change in CSP sub-system ctrl operational state.

This is a callback hook, called by the EventManager when the operational state of a CSP sub-system ctrl changes.

**Parameters**

- **component** (Component) – the sub-system component whose operational state has changed
- **op_state** (*DevState*) – the new operational state of the sub-system ctrl.

**perform_action**(*action: str*) → None

Not operative method.

This method is required by the *CspSubarray* InitCommand class that must inherit from the SKABaseDevice.InitCommand because all the SKA attributes and logging are initialized there.

### Controller Health State Model

**class** ska_csp_lmc_common.controller.controller_health_state.**ControllerHealthModel**(*init_state: ska_tango_base.control_mo health_changed_callback: Callable[[ska_tango_base.c None]*, *logger=None*)

Bases: HealthStateModel

A simple health state model for the CSP Controller that supports.

- HealthState.OK – when the component is fully operative.
- HealthState.DEGRADED – when the component is partially operative.
- HealthState.UNKNOWN – when communication with the component is not established.
- HealthState.FAILED – when the component has faulted

**evaluate_health**() → ska_tango_base.control_model.HealthState

Compute overall health of the controller.

The overall health is based on the fault and communication status of the CSP sub-system controllers.

**Returns**
an overall health of the controller

**Return type**
HealthState

**component_health_changed**(*component: Component*, *health_state: HealthState | None*) → None

Handle change in the health of the CSP subordinate sub-systems controllers.

This is a callback hook, called by the EventManager when the health and/or operational state of one of the CSP subordinate sub-system changes.

**Parameters**

- **component** (Component) – the sub-system component whose health has changed.
- **health_state** (*HealthState*) – the new health state of the CSP sub-system.

### Subarray Operational State Model

**class** ska_csp_lmc_common.subarray.subarray_op_state_model.**SubarrayOpStateModel**(*op_state_init*, *op_state_changed_callback: Callable[[tango.DevState], None]*, *logger=None*)

Bases: OpStateModel

A simple operational state model that supports.

- DevState.ON – when the component is powered on.
- DevState.OFF – when the component is powered off.
- DevState.STANDBY – when the component is low-power mode.
- DevState.ON – when the component is powered on.
- DevState.DISABLE – when the component is in OFFLINE administrative mode.
- DevState.UNKNWON – when communication with the component is not established.
- DevState.FAILED – when the component has faulted

**property op_state:** tango.DevState

Return the operational state.

**Returns**
the operational state state.

**Return type**
DevState

**update_op_state**() → None

Update the operational state.

This method calls the :py:meth:evaluate_op_state method to figure out what the new operational state should be, and then updates the op_state attribute, calling the TANGO device callback if required.

**evaluate_op_state**() → tango.DevState

Compute overall operational state of the CSP subarray device based on the fault and communication status of the subarray overall, together with the aggregation of the operational states of the subordinate sub-systems components.

**Returns**
an overall operational state of the subarray.

**Return type**
DevState

**component_op_state_changed**(*component: Component*, *op_state: DevState | None*) → None

Handle change in the operational state of a CSP sub-system subarray/beam.

This is a callback hook, called by the EventManager when the operational state of a CSP sub-system subarray/beam changes.

**Parameters**

- **component** (Component) – the sub-system component whose operational state has changed
- **op_state** (*DevState*) – the new operational state of the sub-system subarray/beam.

**perform_action**(*action*)

Not operative method.

This method is required by the *CspSubarray* InitCommand class that must inherit from the SKABaseDevice.InitCommand because all the SKA attributes and logging are initialized there.

### Subarray Health State Model

**class** ska_csp_lmc_common.subarray.subarray_health_model.**SubarrayHealthModel**(*init_state: ska_tango_base.control_model.Hea health_changed_callback: Callable[[ska_tango_base.control_ None], logger=None*)

Bases: HealthStateModel

A simple health state model for the CSP Subarray that supports.

- HealthState.OK – when the component is fully operative.
- HealthState.DEGRADED – when the component is partially operative.
- HealthState.UNKNWON – when communication with the component is not established.
- HealthState.FAILED – when the component has faulted

**evaluate_health**() → ska_tango_base.control_model.HealthState

Compute overall health of the subarray.

The overall health is based on the fault and communication status of the subarray overall, together with the health of its stations and subarray beams.

This implementation set the health of the CSP Subarray to the health of CBF Subarray component.

**Returns**
an overall health of the subarray.

**Return type**
HealthState

**component_health_changed**(*component: ObservingComponent*, *health_state: HealthState | None*) → None

Handle change in the health of the CSP Subarray subordinate sub- systems subarras/beams.

This is a callback hook, called by the Event Manager when the health of a CSP Subarray subordinate subsystem changes.

**Parameters**

- **component** (Component) – the sub-system component whose health has changed
- **health_state** (*HealthState*) – the new health state of the sub-system.

### Subarray Observing State Model

**class** ska_csp_lmc_common.subarray.subarray_obs_state_model.**SubarrayObsStateModel**(*obs_state_init*, *obs_state_changed_callback: Callable[[ska_tango_base.co None]*, *logger=None*)

Bases: ObsStateModel

A simple observing state model that supports the Observing State as described in ADR-8.

The subarray observing state at initialization is ObsState.EMPTY. This is the same value reported when communication with the component is not established or unknown.

**evaluate_obs_state**() → ska_tango_base.control_model.ObsState

Compute overall observing state of the CSP Subarray based on the fault and communication status of the subarray overall, together with the aggregation of the operational states of the subordinate sub- systems components. as aggregation of the observing states of the subordinate sub-systems (subarrays/beams).

**Returns**
the overall observing State of the CSP subarray.

**Return type**
ObsState

**component_obs_state_changed**(*component: ObservingComponent*, *obs_state: ObsState | None*) → None

Handle change in the observing state of the CSP sub-system subarrays/beams.

This is a callback hook, called by the EventManager when the observing state of a CSP sub-system subarray/beam changes.

**Parameters**

- **component** – the CSP sub-system component whose observing state has changed
- **obs_state** (*ObsState*) – the new obsState of the sub-system subarray/beam

**Type**
Component

CHAPTER

FOUR

# INDICES AND TABLES

- genindex
- modindex
- search

# PYTHON MODULE INDEX

## S

# INDEX

## Symbols



## M



## S