CSP.LMC Common Software Documentation

Release 0.17.6

SKA Organization

Sep 20, 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)

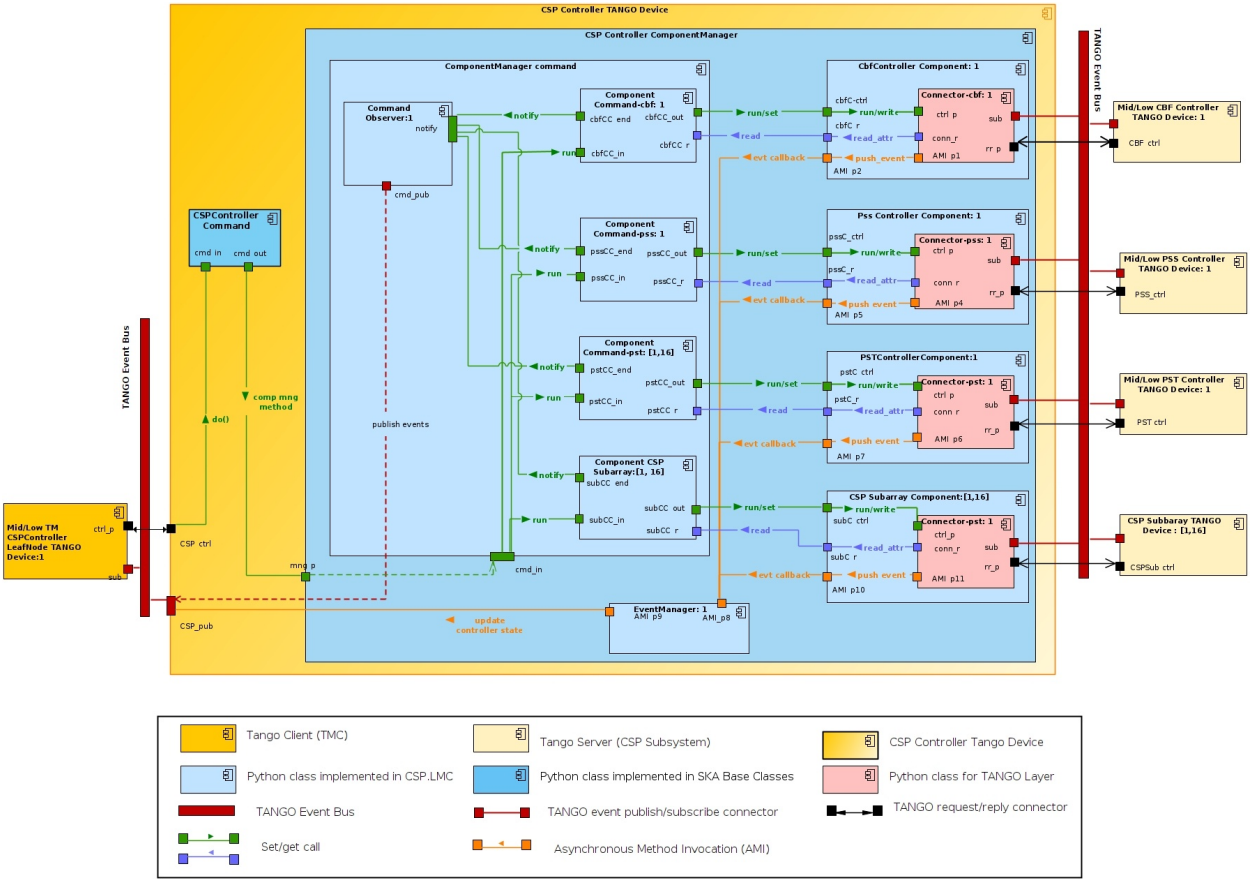
CHAPTER

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

3.1.2 CspSubarray

3.2 CSP.LMC modules API

3.2.1 Manager subpackage

Controller Component Manager

Subarray Component Manager

Event Manager

Component Manager Configurator

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: CspEvent*)

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 subscription, 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_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_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

class ska_csp_lmc_common.observing_component.**ObservingComponent**(*fqdn: str*, *name: str*, *weight: int = 0*, *logger: Optional[Logger] = None*)

Bases: Component

Class to model a CSP subordinate observing device.

property capability_id: int

Return the capability device identification number.

Returns
The capability device ID.

property obs_state: ska_control_model.ObsState

Return the CSP Subarray sub-system obs_state.

Returns
the sub-system obsState if updated via events or via direct read, EMPTY on failure

set_tmp_command_name(*value: str*)

Set a temporary variable to modify the name of the command. It is needed since different version of base classes have different command name (from low cbf (> 0.6.1))

Parameters
value – the value of the command name that have to be sent to the subarray

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 not changed.

set_component_unknown(*admin_mode_value: ska_control_model.AdminMode*) → None

Specialized version for observing sub-system components.

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

Returns
None

set_component_offline(*admin_mode_value: ska_control_model.AdminMode*) → None

Specialized version for observing sub-system components.

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

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

Run the command configure scan on the component.

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

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

Run the command gotoidle on the component.

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

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

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

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 from observing_component

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

Specialize the command gotoidle from observing_component.

PSS Subarray Component

PST Beam Component

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.

_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**(*component: 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**(*component: 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**(*component: 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**(*component: 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**(*component: 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_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.

succeeded() → None

This command does not specialize the *succeeded* method.

failure_detected() → None

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.

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.

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]*, *timeout: Optional[int] = 0*, *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]*, *timeout: Optional[int] = 0*, *logger: Optional[Logger] = None*, *abort_event: Optional[Event] = None*)

Parameters

- **name** – the name of the macro command.
- **command_factory** – the factory class to create the CSP Component commands classes.
- **notify_callback** – the observer method invoked at command completion.
- **macro_items** – a list of MacroCommandItem entries
- **logger** – the device logger target
- **abort_event** – the event set on abort request.

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.

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.

Health State Model

class ska_csp_lmc_common.model.**HealthStateModel**(*init_state: ska_control_model.HealthState*, *health_changed_callback: Callable[[ska_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_control_model.HealthState*, *health_changed_callback: Callable[[ska_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_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_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_control_model.ObsState, obs_state_changed_callback: Callable[[ska_control_model.ObsState], None], logger: Optional[Logger] = None*)

Bases: `object`

A simple observing state model for observing devices.

`__init__`(*obs_state_init: ska_control_model.ObsState, obs_state_changed_callback: Callable[[ska_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_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_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 administrative 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.

perform_action(*action: str*) → None

Perform action to trigger the obs-state-model To be implemented

Parameters
action – The action to perform: *invoked* or *completed*

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.

Controller Health State Model

class ska_csp_lmc_common.controller.controller_health_state.**ControllerHealthModel**(*init_state: ska_control_model.HealthS health_changed_callback: Callable[[ska_control_mod None], log- ger=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_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

Subarray Health State Model

Subarray Observing State Model

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

INDEX

Symbols

M

S