MID CSP.LMC Software Documentation

Release 0.16.2

SKA Organization

Sep 20, 2023

MID.CSP LMC

CHAPTER

ONE

MID.CSP LMC DOCUMENTATION

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

- *Mid.CSP LMC Controller*
- *Mid.CSP LMC Subarray*
- Mid.CSP Alarm Handler
- Mid.CSP Logger
- Mid.CSP TANGO Facility Database
- VCC Capability (receptors)
- *FSP Processing Modes Capability*
- *Search Beam Capability*
- *Timing Beam Capability*
- *VLBI Beam Capability*

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*.

1.1 Mid.CSP LMC Controller

Mid.CSP Controller is the top-level TANGO Device and the primary point of contact for monitor and control of the Mid.CSP Sub-syste.

The Mid.CSP Controller represents Mid.CSP Sub-system as a unit for control and monitoring for general operations.

Mid.CSP Controller main roles are:

- To be the central control node for Mid.CSP. The Controller provides a single point of access for control of the Mid.CSP as a whole, this includes provision for housekeeping and supervisory commands including: power-up, power-down, power management, restart (re-initialize), support for firmware and software upgrades, etc.
- To provide rolled-up reporting for the overall Mid.CSP status. Mid.CSP Controller monitors and intelligently rolls-up status reported by Mid.CSP equipment, sub-arrays and capabilities and maintains a standard set of states and modes, as defined in the document "SKA1 Control System Guidelines". State transitions are reported using standard TANGO mechanism.
- To implement a set of attributes that represent the status and configuration of the Mid.CSP as a whole and, where required, report availability, status and configuration of the Mid.CSP equipment, components and capabilities (in the form of lists or tables or JSON string).

- To maintains the pool of resources (VCCs, FSPs, Search/Timing/VLBI beams), keep track of allocation to sub-arrays and provide reports on resource availability, allocation, and more.

Mid.CSP Controller implementation is based on (derived from) the standard SKA1 TANGO Device Controller; Mid.CSP Controller implements the standard TANGO API, aligned with the document "SKA1 Control System Guidelines".

The interface is between a TANGO client and a TANGO Device. The TANGO Device exposes attributes and commands to clients.

The roles of the interfacing systems are:

- TANGO Clients: Mid.TMC sub-system TANGO client(s).
- TANGO Device: Mid.CSP sub-system implements Mid.CSP Controller.

The clients use requests to obtain read and/or write access to TANGO device attributes, and to invoke TANGO device commands.

1.1.1 Mid.CSP Controller TANGO Device name

The Mid.CSP Controller TANGO Device name is defined in the document "SKA1 TANGO Naming Conventions":

```
mid-csp/control/0
```

1.1.2 Mid.CSP Controller TANGO Device Properties

The Mid.CSP Controller device has a standard set of properties inherited from the SKA Controller TANGO Device and a number of specific properties documented in the Controller API section.

1.1.3 Mid.CSP Controller TANGO Device States and Modes

Mid.CSP Controller implements the standard set of state and mode attributes defined by the SKA Control Model.

Mid.CSP Controller reports on behalf of the Mid.CSP Sub-system – unless explicitly stated otherwise, the state and mode attributes implemented by the Mid.CSP Controller represent the status of the Mid.CSP as a whole, not the status of the Mid.CSP Controller itself.

1.1.4 Mid.CSP Controller operational state

The Mid.CSP Controller supports the following sub-set of the TANGO Device states:

- UNKNOWN: Mid.CSP is unresponsive, e.g. due to communication loss. This state cannot be reported by CSP itself.
- OFF: power is disconnected. This state cannot be reported by CSP itself
- INIT: Initialization of the monitor and control network,equipment and functionality is being performed. During initialization commands that request state transition to OFF (power-down) or re-start initialization are accepted.
- DISABLE: Mid.CSP is administratively disabled, either by setting adminMode=OFFLINE or NOT-FITTED. Basic monitor and control functionality is available but signal processing functionality and related commands are not available. All sub-arrays are empty (OFF) and IDLE; all resources (receptors, tide-array beams) are placed in the pool of unused resources.)

- STANDBY: Low-power state, Mid.CSP uses < 5% of nominal power. Basic monitor and control functionality is available, including the commands to request state transition to ON, OFF, DISABLE, or INIT. Signal processing functionality and related commands are not available. All sub-arrays are empty (OFF) and IDLE; all resources (receptors, tide-array beams) are placed in the pool of unused resources.).
- ON: At least a minimum of CSP signal processing capability is available; at least one receptor and one sub-array can be used for observing (either for scientific observations or for testing andmaintenance). Mid.CSP is in normal operational state, all commands, including commands to increase/decrease functional availability and power consumption are available.
- ALARM: Quality Factor for at least one attribute crossed the ALARM threshold. Part of Mid.CSP functionality may be unavailable.
- FAULT: Unrecoverable fault has been detected, Mid.CSP is not available for use at all, maintainer/operator intervention is required in order to return to ON, STANDBY, or DISABLE. Depending on the extent of failure commands restart and init, as well as status reporting may be available.

1.1.5 Mid.CSP Controller TANGO Device Commands

Mid.CSP Controller implements the standard set of commands as specified in:

- Standard set of TANGO Device commands as defined in TANGO User Manual
- Standard set of SKA TANGO Device commands
- Command specific to Mid.CSP Controller as described in API section.

Mid.CSP makes provision for TM to request state transitions for individual sub-systems and/or Capabilities.

1.1.6 Mid.CSP Controller TANGO Device Attributes

Mid.CSP Controller implements the standard set of attributes as specified in:

- Standard set of TANGO Device attributes as defined in TANGO User Manual
- The standard set of SKA TANGO Device attributes as defined for the SKA Controller TANGO Device.
- Attributes to Mid.CSP Controller as described in API section.

The Mid.CSP Controller maintains the 'pool of resources' and is able to provide information regarding sub-array membership, status and usage.

1.2 Mid.CSP LMC Capabilities

The Mid CSP.LMC Capabilities provide a layer of abstraction to allow the TM to set, control and monitor signal processing, and other sub-system functions, without being aware of the details of the Mid CSP implementation.

Each Mid CSP.LMC Capability represents a subset of the Mid.CSP functionality and uses a subset of the Mid.CSP hardware and software resources.

- VCCs (Very Coarse Channelizer): 197
- FSPs (Frequency Slice Processor): 27
- PSS (Pulsar Search Beams): 1500.
- PST (Pulsar Timing Beams): 16

The Capabilities implemented by the Mid.CSP can be classified in two groups, as follows:

- The Capabilities inherent to the sub-arrays or other Capabilities; such Capabilities are created during initialization and are permanently assigned to a particular sub-array or Capability. A Capability can be enabled/disabled, but cannot be removed and/or assigned to another subarray.
- The Capabilities that are function of resources: these are created during initialization and assigned to a pool of unused resources. Individual instances of these resources may be assigned to subarrays, as commanded by the TM. In some cases the same instance may be used in a shared manner by more than one subarray.

Each Mid.CSP Subarray has the following inherent Capabilities that correspond to Processing Modes, and are configured via scan configuration:

1. Correlation,
2. PSS,
3. PST,
4. VLBI (beamforming and correlation in support of VLBI), and
5. Transient Data Capture.

The Capabilities listed above are configured and controlled via a subarray; when a subarray does not perform correlation, its Capability Correlation is OFF; the same applies for all Processing Modes.

The top-level Mid.CSP Capabilities associated to schedulable resources are reported in the table below:

Capability Name	Resources	Notes
Frequency Observing Band	197 VCCs (Very Coarse Channeliser)	Support frequency bands: • Band 1 • Band 2 • Band 3 • Band 4 • Band 5a • Band 5b
FSP Processing Modes	27 FSPs (Frequency Slice Processor) These resources are shareable among the subarrays	Supported processing modes: • correlation • pulsar search beamforming • pulsar timing beamforming
PSS Processing Mode	1500 Search Beams	Collects the HW, SW and processing modes of PSS: beams: This capability is a func of the following resources: • receptors • CBF beamformers • PSS pipelines • SDP links
PST Processing Mode	16 Timing Beams	Collects the HW, SW and processing modes of PST: beams: This capability is a func of the following resources: • receptors • CBF beamformers • PST processors • SDP links

The previous list does not include VLBI. VLBI beamforming is configured per subarray per FSP, any further beam processing is performed by non-SKA1 equipment; there are no VLBI related Mid.CSP resources that have to be managed, other than the FSPs.

1.3 Mid.CSP LMC Subarray

The Mid.CSP Sub-array TANGO Device class is required as a common interface feature, as described in the document "SKA1 Control System Guidelines".

Its purpose is to provide a point of access for configuration, execution and monitoring of signal processing functionality for each sub-array independently.

1.3.1 Overview of the functionality

1. Add/release resources (receptors, beams) to/from the sub-array. TM accesses directly the Mid.CSP Sub-array in order to add/release resources to/from sub-arrays. The TM requests are validated by the Mid.CSP sub-array relying on the information maintained by the Mid.CSP Controller and Capability.
2. Set the frequency Band.
3. Configure the sub-array Processing Modes using command *Configure()*. This includes configuration of the Capabilities used by the sub-array.
4. Start/stop scan execution.
5. Set 'engineering' parameters for the sub-arrays and Capabilities.

1.3.2 Receptors assignment

The assignment of receptors to a MID sub-array is performed in advance of a scan configuration.

Up to 197 receptors can be assigned to one sub-array.

Receptors assignment to a sub-array is exclusive: one receptor can be assigned only to one sub-array.

During assignment, the Mid.CSP sub-array performs both formal and more deep checks.

Formal control: - The assigned receptor list is empty. In this case, a warning is logged. - The receptor names are compliant to ADR-32. In this case, a warning is logged for each receptor not compliant and the receptor is excluded from the assigned list. - The assigned receptor list contains duplicated receptors. In this case, the duplicated receptor is removed from the list and a warning is logged.

Deep control: - The requested receptors are in the list of the deployed ones. - the requested receptors are already associated to another sub-array. In this case a warning message is logged.

In order for a Mid.CSP sub-array to accept a scan configuration, at least one receptor must be assigned to the sub-array.

1.3.3 Tied-array beams

Depending on the selected processing mode, Search Beams, Timing Beams and Vlbi Beams must be assigned to a MID sub-array in advance of a scan.

TM may specify the number of beams to be used or, alternatively, identify the Capability instances to be used via their TANGO Fully Qualified Domain Name (FQDN).

1.3.4 Scan configuration

TM provides a complete scan configuration to a sub-array via an ASCII JSON encoded string.

When a complete and coherent scan configuration is received and the sub-array configuration (or re-configuration) completed, the sub-array it's ready to observe.

Once configured, Mid.CSP keeps the sub-array and scan configuration until one of the following occurs:

- Mid.CSP receives a command to release resources (receptors, FSPs, Search Beams and/or Timing Beams) used by the sub-array.
- Mid.CSP receives a command to transition to low-power state (state=STANDBY); in which case the active scans end and resources are released.
- Mid.CSP receives a command to shut-down (OFF)
- Mid.CSP monitor and control function fails so that the configuration is lost.

Inherent Capabilities

The following inherent Capabilities that correspond to Processing Modes, and are configured via scan configuration:

1. Correlation
2. PSS
3. PST
4. VLBI (beamforming and correlation in support of VLBI)
5. Transient Data Capture.

When a sub-array does not perform correlation, its Capability Correlation is OFF; the same applies for all Processing Modes.

1.3.5 Control and Monitoring

Each Mid.CSP Sub-array maintains and report the status and state transitions for the Mid.CSP sub-array as a whole and for the individual assigned resources.

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

1.3.6 Mid.CSP Sub-array TANGO Device name

The Mid.CSP Sub-array TANGO Device name is defined in the document "SKA1 TANGO Naming Conventions":

```
mid-csp/subarray/XY
```

where XY is a two digit number in range [01,..,16].

1.3.7 Mid.CSP Sub-array operational state

Mid.CSP Sub-array intelligently rolls-up the operational state of all components used by the sub-array and reports the overall operational state for the sub-array.

The Mid.CSP Sub-array supports the following sub-set of the TANGO Device states:

- UNKNOWN: Mid.CSP sub-array is unresponsive, e.g. due to communication loss.
- OFF: The sub-array is not enabled to perform signal processing. The sub-array is 'empty'.
- INIT: Initialization of the monitor and control network,equipment and functionality is being performed.
- DISABLE: Mid.CSP sub-array is administratively disabled, basic monitor and control functionality is available but signal processing functionality is not available.
- ON: The sub-array is enabled to perform signal processing. The sub-array observing state is EMPTY if receptors have not been assigned to the sub-array, yet.
- ALARM: Quality Factor for at least one attribute crossed the ALARM threshold. Part of functionality may be unavailable.
- FAULT: Unrecoverable fault has been detected. The sub-array is not available for use and maintainer/operator intervention might be required.

1.3.8 Mid.CSP Sub-array observing state

The sub-array Observing State indicates status related to scan configuration and execution.

The Mid.CSP Sub-array observing state adheres to the State Machine defined by ADR-8.

1.3.9 Mid.CSP Sub-array TANGO Device Commands

Mid.CSP Sub-array implements the standard set of commands as specified in:

- Standard set of TANGO Device commands as defined in TANGO User Manual
- Standard set of SKA TANGO Device commands
- Command specific to Mid.CSP Sub-array as described in API section.

Mid.CSP makes provision for TM to request state transitions for individual sub-systems and/or Capabilities.

1.3.10 Mid.CSP Sub-array TANGO Device Attributes

Mid.CSP Sub-array implements the standard set of attributes as specified in:

- Standard set of TANGO Device attributes as defined in TANGO User Manual
- The standard set of SKA TANGO Device attributes as defined for the SKA Sub-array TANGO Device.
- Attributes to Mid.CSP Sub-array as described in API section.

Virtually all parameters provided in scan configuration are exposed as attributes either by Mid.CSP Sub-array or by individual Capabilities.

CHAPTER

TWO

MID CSP.LMC ARCHITECTURE

The architecture of the CSP.LMC software is the same for Low and Mid Telescope. Please refer to common documentation.

CHAPTER

THREE

PROJECT'S API

Below are presented the API for Classes specialized for Mid Telescope.

All the functionalities common to Mid and Low Telescope are developed in the project ska-csp-lmc-common

3.1 Mid.CSP LMC Devices API

3.1.1 Mid CSP.LMC Controller

class ska_csp_lmc_mid.mid_controller_device.**MidCspController**(**args: Any*, ***kwargs: Any*)

Bases: CspController

The base class for MID CspMAster. Functionality to monitor CSP.LMC Capabilities are implemented in separate TANGO Devices.

Device Properties:

CspVccCapability

- TANGO Device to monitor the Mid.CSP VCCs Capabilities devices.
- Type:'DevString'Properties

CspFspCapability

- TANGO Device to monitor the Mid.CSP FSPs Capabilities devices.
- Type:'DevString'

set_component_manager(*cm_configuration: ska_csp_lmc_common.manager.manager_configuration.ComponentManagerConfiguration*) → MidCspControllerComponentManager

Set the Component Manager for the Mid CSP Controller device.

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

Returns
The Mid.Csp ControllerComponentManager

commandResultName() → str

Return the name of the last executed CSP task.

Returns
The name of the CSP task.

commandResultCode() → str

Return the *ResultCode* of the last executed CSP task.

Returns
The result code (see ResultCode) of the last executed task.

availableCapabilities() → List[str]

Return the list of available capabilities.

Returns
For each Mid.CSP capability type, reports the number of available resources.

receptorsList() → List[str]

Return the list with all the (deployed) receptors IDs.

Returns
The list of receptors Ids.

receptorMembership() → List[int]

Return the information about the receptors/VCCs affiliation to Mid.CSP subarrays. The value stored in the (receptor_id -1) element corresponds to the subarray Id owing the VCC associated to the receptor. This value is in the range [0, 16]: 0 means the current resource is not assigned to any subarray.

On failure an empty list is returned (VERIFY!!!)

Returns
The subarray affiliation of the receptors.

unassignedReceptorIDs() → List[str]

Return the list of available receptors IDs. The list includes all the receptors that are not assigned to any subarray and, are "full working". This means:

- a valid link connection receptor-VCC
- the connected VCC healthState OK

On failure an empty list is returned.

Returns
The list of the available receptors IDs.

cspVccCapabilityAddress() → List[str]

Return the CSP VCC Capability device address.

cspFspCapabilityAddress() → List[str]

Return the CSP FSP Capability device address.

vccAddresses() → List[str]

ReturnCspFspCapabilityVcc the list with the FQDNs of the VCCs TANGO devices.

Returns
the list of VCC Capabilities FQDNs

fspAddresses() → List[str]

Return the list with the FQDNs of the FSPs TANGO devices.

Returns
The list of FSP Capabilities FQDNs

3.1.2 MID CSP Subarray

class ska_csp_lmc_mid.mid_subarray_device.**MidCspSubarray**(**args: Any*, ***kwargs: Any*)

Bases: CspSubarray

The base class for MID CspSubarray.

Functionality to monitor assigned CSP.LMC Capabilities, as well as inherent Capabilities, are implemented in separate TANGO Devices.

set_component_manager(*cm_configuration: ska_csp_lmc_common.manager.manager_configuration.ComponentManagerConfiguration*) → MidCspSubarrayComponentManager

Set the CM for the Mid CSP Subarray device.

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

Returns
The Mid.CSP SubarrayComponentManager

read_commandResultName() → str

Return the name of the last executed CSP task.

Returns
The name of the CSP task.

read_commandResultCode() → str

Return the *ResultCode* of the last executed CSP task.

Returns
The result code (see ResultCode) of the last executed task.

read_addReceptorsCmdProgress() → int

Return the addReceptorsCmdProgress attribute.

Returns
The progress percentage for the AddReceptors command.

NOTE: not implemented

read_removeReceptorsCmdProgress() → int

Return the removeReceptorsCmdProgress attribute.

Returns
The progress percentage for the RemoveReceptors command.

NOTE: not implemented

read_assignedFsp() → Tuple[int]

Return the assignedFsp attribute.

Returns
List of assigned FSPs.

NOTE: not implemented

read_assignedVcc() → List[int]

Attribute method

Returns
The list of VCC IDs assigned to the subarray.

read_assignedVccState() → List[tango.DevState]

Return the assignedVccState attribute.

Returns
The State of the assigned VCCs.

read_assignedVccHealthState() → List[ska_control_model.HealthState]

Return the assignedVccHealthState attribute.

Returns
The health state of the assigned VCCs.

read_assignedFspState() → Tuple[tango.DevState]

Return the assignedFspState attribute.

Returns
The State of the assigned FSPs.

NOTE: not implemented

read_assignedFspHealthState() → Tuple[int]

Return the assignedFspHealthState attribute.

Returns
The health state of the assigned FSPs.

read_assignedReceptors() → List[str]

Return the assignedReceptors attribute.

Returns
The list of receptors assigned to the subarray.

3.1.3 MID CSP.LMC FSP Processing Mode Capability Device

class ska_csp_lmc_mid.mid_capability_fsp_device.**MidCspCapabilityFsp**(**args: Any*, ***kwargs: Any*)

Bases: SKABaseDevice

Mid CSP Capability FSP device Aggregates FSP information and presents to higher level devices

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

Bases: InitCommand

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

do()

Stateless hook for device initialisation.

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

Return type
(ResultCode, str)

init_device()

Initialise the tango device after startup.

set_component_manager(*cm_configuration: ska_csp_lmc_common.manager.manager_configuration.ComponentManagerConfiguration*) → MidFspCapabilityComponentManager

Configure the ComponentManager for the CSP Capability FSP device.

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

Returns
The CSP Capability FSP ComponentManager

create_component_manager() → MidFspCapabilityComponentManager

Override the base method.

Returns
The CSP ControllerComponentManager

update_device_attribute(*attr_name: str*, *attr_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

- **attr_name** – the TANGO attribute name
- **attr_value** – the attribute value

fspsDeployed() → int

Read number of deployed FSPs

fspFqdn() → List[str]

Read list of fqdn for FSPs

fspFunctionMode() → List[str]

Read list of function mode for FSPs

fspState() → List[tango.DevState]

Read list of state for FSPs

fspHealthState() → List[str]

Read list of health state for FSPs

fspAdminMode() → List[str]

Read list of admin mode for FSPs

fspAvailable() → List[int]

Read list of available FSPs IDs, meaning function_mode=idle and state=ON

fspUnavailable() → List[int]

Read list of unavailable FSPs IDs, meaning state!=ON

fspPss() → List[int]

Read list of FSPs IDs which are in function_mode=PSS

fspPst() → List[int]

Read list of FSPs IDs which are in function_mode=PST

fspCorrelation() → List[int]

Read list of FSPs IDs which are in function_mode=Correlation

fspsSubarrayMembership() → List[str]

Return the FSPs subarray membership as a List of strings. Each string contains the IDs of the subarrays to which a FSP belongs to.

fspsFqdnAssigned()

Return the FQDNs of the FSPs assigned to subarrays.

The value is a matrix of dim 16*4 where in each row is reported the list of the FQDNs of the FSPs belonging to the subarray whose id = nrow * 1

fspJson() → str

Read json which contain raw FSP information.

isCommunicating() → bool

Whether the TANGO device is communicating with the controlled component.

adminMode(*value: ska_control_model.AdminMode*) → None

Set the Admin Mode of the device.

Parameters
value – Admin Mode of the device.

Raises
ValueError – for unknown adminMode

fspsSwVersion() → List[str]

Return the online fsps software Version attribute list.

Returns
the online fsps software Version attribute list.

fspsHwVersion() → List[str]

Return the online fsps hardware Version attribute list.

Returns
the online fsps hardware Version attribute list.

fspsFwVersion() → List[str]

Return the online fsps firmare Version attribute list.

Returns
the online fsps firmware Version attribute list.

3.2 Mid.CSP LMC modules API

3.2.1 Manager subpackage

FSP Processing Mode Component Manager

class ska_csp_lmc_mid.manager.mid_fsp_capability_component_manager.**MidFspCapabilityComponentManager**(**arg*
Any,
***kw*
Any)

Bases: CSPBaseComponentManager

Class for Mid Csp Fsp Capability Component Manager.

__init__(*properties: ska_csp_lmc_common.manager.manager_configuration.ComponentManagerConfiguration, max_workers: Optional[int] = 5, update_device_property_cbk: Optional[Callable] = None, logger: Optional[Logger] = None*) → None

Initialize the FspCapability Component Manager Class

Parameters

- **properties** – A class instance whose properties are the device properties.
- **max_workers** – the number of worker threads used by the ComponentManager ThreadPool
- **update_device_property_cbk** – The device method invoked to update the attributes. Defaults to None.
- **logger** – The device or python logger if default is None.

property fsp_fqdn: List[str]

Return the list of FSPs FQDN.

property fsp_state: List[str]

Return the list of FSPs state.

property fsp_health_state: List[str]

Return the list of FSPs health state.

property fsp_admin_mode: List[str]

Return the list of FSPs admin mode.

property fsp_function_mode: List[str]

Return the list of FSPs function modes.

property fsp_json: str

Return a JSON string with the overall FSPs status.

Returns

the serialized JSON string otherwise an empty string.

property fsp_available: List[int]

Return the list of available FSPs. A FSP is available when its state is ON and the functionMode is IDLE.

property fsp_correlation: List[int]

Return the list of FSPs programmed for correlation.

property fsp_pss: List[int]

Return the list of FSPs programmed for pss beam-forming.

property fsp_pst: List[int]

Return the list of FSPs programmed for pst beam-forming.

property fsp_unavailable: List[int]

Return the list of FSPs not available. A FSP is not available when its state is not ON.

property fsps_deployed: int

Return the number of FSPs deployed in the system.

property fsps_fw_version: str

Return the list of the FSPs online firmware version.

property fsps_sw_version: str

Return the list of the FSPs online software version.

property fsps_hw_version: str

Return the list of the FSPs online hardware version.

property fsps_subarray_membership: list[str]

Return the FSPs subarray membership.

Returns
a list of strings. Each string reports the subarray IDs to which a FSP belongs to.

Note: Example:

fsp_num = list_idx + 1

[“1,2”, “1,3,4”, “”, “”]

FSP 1 is assigned to subarrays 1 and 2 FSP 2 is assigned to subarrays 1,3,4 The other FSPs are not assigned at all.

---

property fsps_fqdn_assigned: List[List[str]]

Return the nested list of the FSPs’ fqdn assigned to all subarrays.

Note: Examples: fsps_fqdn_assigned = [

[‘mid_csp_cbf/fsp/01’, ‘’,”,”], [‘’, ‘mid_csp_cbf/fsp/03’,”,”], [‘’, ‘’,”,”], [‘mid_csp_cbf/fsp/02’,’mid_csp_cbf/fsp/02’,”,”], ….

]

fsps_fqdn_assigned[0] = FQDNs of the FSPs assigned to subarray1 fsps_fqdn_assigned[1] = FQDNs of the FSPs assigned to subarray2 …..

_connect_to_cbf_controller() → List[str]

Instantiate the connection with the Mid CBF Controller.

The Mid CBF Controller FQDN is defined as a device property as well as the connection timeout and polling time.

Device property: CspCbf
ConnectionTimeout PingConnectionTime

_handler_to_invoke(*status: Optional[TaskStatus] = None*, *result: Optional[tuple[ResultCode, str]] = None*)

Callbk invoked when the connection with all the FSP devices is completed.

Parameters

- **task_name** – the name of the executed task
- **result_code** – the task result
- **command_id** – the unique task id, if any
- **task_status** – the task status code, if any
- **result_msg** – the message associated with the task result code.

_read_fsp_version(*fsp_online*) → None

Read the Software, firmware and hardware version attributes
from an online FSP.

Parameters
fsp_online – one of the online FSP

_connect_to_fsp_devices(*list_of_fsp_devices*, *task_callback: Optional[Callable] = None*) → None

Perform the connection with the FSP devices.

Instantiate the command observer and an init command for each FSP component. Each connection is executed in a separate worker thread.

Parameters

- **list_of_fsp_devices** – the list of CBF FSP devices FQDNs
- **task_callback** – registered method invoked when task ends.

start_communicating() → None

Start the connection with the Mid CBF Controller device to retrieve the list of the MId.CBF FSP Capability devices FQDNs.

This method is executed asynchronously

off(*task_callback: Optional[Callable] = None*)

Method not used by this device

on(*task_callback: Optional[Callable] = None*)

Method not used by this device

standby(*task_callback: Optional[Callable] = None*)

Method not used by this device

stop_communicating(*task_callback: Optional[Callable] = None*)

Method not used by this device

abort_commands(*task_callback: Optional[Callable] = None*)

Method not used by this device

class ska_csp_lmc_mid.manager.mid_fsp_capability_component_manager.**FspInfoStruct**(*task_callback: Optional[Callable] = None*, *logger: Optional[Logger] = None*)

Bases: object

Class handling the update and access to the dictionary with the overall FSPs status.

__init__(*task_callback: Optional[Callable] = None*, *logger: Optional[Logger] = None*) → None

Class to handle the FSPs information.

Parameters

- **task_callback** – the method invoked to update a device attribute
- **logger** – The device or python logger if default is None.

property fsp_json_dict: Dict

Return the internal dictionary.

property unavailable: List[int]

Return the list of the FSPs that are not available.

property fqdn: List[str]

Return a list with the FSPs FQDNs.

init(*list_of_fsp_devices*)

Initialize the internal FSP json dictionary.

update(*device_fqdn: str*, *attr_name: str*, *attr_value: Any*) → None

Method invoked when an event is received.

Update the FSP internal json structure and invoke the device update method for the following attributes of the device: - fspDeployed - fspFunctionMode - fspAdminMode - fspHealthState - fspState

Parameters

- **device_fqdn** – the FSP device FQDN
- **attr_name** – the name of the attribute with event
- **attr_value** – the value the attribute with event

fsp_in_function_mode(*function_mode*) → List[int]

Return the list of the FSP id that are in the requested function mode.

Parameters
function_mode – the requested functionMode (label)

get_fsp_values(*dict_key*)

Return the information stored in the dictionary for the given key.

Parameters
dict_key – the dictionary key whose value is requested.

_update_functionmode() → None

Update the list with the FPSs functionMode.

This method is invoked when an event on the functionMode attribute is received.

_update_state() → None

Update the list with the FPSs state.

This method is invoked when an event on the State attribute is received.

_update_healthstate() → None

Update the list with the FPSs health state.

This method is invoked when an event on the healthState attribute is received.

_update_adminmode() → None

Update the list with the FPSs admin mode.

This method is invoked when an event on the adminMode attribute is received.

_update_subarraymembership() → None

Update the list with the FPSs sub-array membership

3.2.2 Mid.CSP Sub-system Components

Components class work as adaptors and caches towards the Mid.CSP sub-systems TANGO devices.

Mid.CBF Controller Component

class ska_csp_lmc_mid.controller.mid_ctrl_component.**MidCbfControllerComponent**(**args: Any*, ***kwargs: Any*)

Bases: CbfControllerComponent

Specialize the CBF Controller Component class for the Mid.CSP LMC.

This class works as a cache and adaptor towards the real Mid device.

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

Initialize the MidCbfControllerComponent.

Parameters

- **fqdn** – The Mid.CBF controller TANGO Device Fully Qualified Domain Name.
- **logger** – The device or python logger if default is None.

property vcc_state: List[tango.DevState]

Return the operational state (State) of the VCCs organized as a list where the element index corresponds to the (vcc_id -1).

Example:

```
vcc_state = [ON, ON, FAULT, UNKNOWN]

VCC 1 -> ON
VCC 2 -> ON
VCC 3 -> FAULT
VCC 4 -> UNKNOWN
```

Returns

a list with the VCCs' healthState.

property vcc_health: List[ska_control_model.HealthState]

Return the healthState of the VCCs organized as a list where the element index corresponds to the (vcc_id -1).

Example:

```
vcc_health = [OK, OK, FAILED, UNKNOWN]

VCC 1 -> OK
VCC 2 -> OK
VCC 3 -> FAILED
VCC 4 -> UNKNOWN
```

Returns

a list with the VCCs' healthState.

property vcc_affiliation: List[int]

Return the affiliation of the VCCs to the Mid.CSP subarrays, organized as a list where.

the position of the element is the receptor_id - 1 and the value is the subarray ID owing that resource. 0 means the current resource is not assigned, yet

Example:

```
vcc_affiliation = [1,0,0,2]

VCC 1 -> subarray 1
VCC 2 -> not assigned
VCC 3 -> not assigned
VCC 4 -> subarray 2
```

Returns

A list with the receptors' affiliation to subarrays

property vcc_to_receptor_map: Dict[str, str]

Return the map vcc-id -receptor-id for all the usable (deployed) receptors as a dictionary.

The format of the attribute is the following one:

```
{vcc_id: receptor_id,...}

Example:

vcc_to_receptor_map = {"1":"SKA002", "2":"SKA004",
                       "3":"SKA001", "4":"SKA003"
                       }
```

Returns

the associations between vcc_id and receptors_id organized as a dictionary.

property vcc_to_receptor: List[str]

Return the value of the Mid.CBF Controller *vccToReceptor* attribute storing a list with the map of all (vcc_id:receptor_index) associations (ordered by vcc id). The receptor index is the (array index + 1) of a full list of 197 receptors. The format of the attribute is the following one:

['vcc_id:receptor_index',...]

Example:

vcc_to_receptor = ['1:2', '2:1', '3:3', '4:75']

Returns

the list with the associations (vcc_id:receptor_index) for each deployed VCC.

property receptor_to_vcc: List[str]

Return the value of the Mid.CBF Controller *receptorToVcc* attribute storing a list with the map of all (receptor_id:vcc_id) associations. (ordered by receptor id). The receptor index is the (array index + 1) of a full list of 197 receptors. The format of the attribute is the following one:

```
['receptor_id:vcc_id',...]

Example:
```

(continues on next page)

(continued from previous page)

```
receptor_to_vcc = ['1:2', '2:1', '3:3', '4:75']
```

Returns
the list with the associations (receptor_index:vcc_id) for each deployed VCC.

property list_of_receptors: List[str]

Return the ordered list of all the usable (deployed) receptors IDs.

Example:

```
If there are only 4 VCC devices deployed:

list_of_receptors = ['SKA001','SKA022','SKA103','MKT002']
```

property linked_receptors: Dict[str, str]

Return information about the receptors IDs with a valid connection to the VCC. Valid receptor ids are in [SKA001,..,SKA133] and [MKT000,MKT63] range. A receptor_id = DIDINV means the link connection between the receptor and the VCC is off. NOTE: Right now, the receptor ID contains always a value, since it is the theoretical connection coming from the TM. In order to be able to check the receptor ID vs VCC link (receptor_id != "DIDINV"), it should be needed to verify the VCC capability attributes.

note:

```
Example:

    list_of_receptors = ["SKA001","SKA002","SKA003","SKA004"]
    vcc_state = [ON, UNKNOWN, ON, ON]
    vcc_to_receptor_map = {'1':"SKA002", '2':"SKA004",
                           '3':"SKA001", '4':"SKA003"
                          }
    linked_receptors = ['1':"SKA002", '3':"SKA001"]
```

Returns
the map of the valid (vcc_id, receptor_id) associations.

property unassigned_receptors: List[str]

Return the not ordered list of availbale receptors IDs. The list includes all the receptors that are not assigned to any subarray and, from the side of CSP, are considered "full working". This means:

- a valid link connection receptor-VCC
- the connected VCC healthState OK

Example:

```
linked_receptors = ["1":"SKA002", "3":"SKA001"]
vcc_affiliation = [1, 0, 0, 3]
unassigned_receptors = ["SKA001"]
```

TODO: Check which is the criteria to establish when a VCC is available
(working): check the healthState (OK) and the state (ON)?

Returns
The list of the unassigned (available) receptors IDs on success, otherwise an empty list.

Raise
ValueError if an error is caught during CBF attributes reading.

property unassigned_vcc: List[str]

Return the list of availbale VCC IDs.

Returns
The list of the unassigned (available) VCC IDs on success, otherwise an empty list.

property receptors_affiliation: List[int]

Build the list reporting the receptors affiliation to subarray, if any. The element index corresponds to (receptor_id -1) and the value stored is the number of the subarray to which the receptor belongs.

Example:

```
vcc_affiliation = [subid=1, subid=2,  ... ]
vcctoReceptor_map = [vccId1:SKA035, vccId2:SKA054]
receptor_list = [SKA001, SKA035,SKA054,...]
receptors_affiliation = [0,1,2,0]

receptors_affiliation = [1,0,0,2]

receptor 1 -> subarray 1
receptor 2 -> not assigned
receptor 3 -> not assigned
receptor 4 -> subarray 2
```

Returns
A list with the receptors' affiliation to subarrays

Returns
The receptors affiliation to subarray_state on success, otherwise an empty list.

get_vcc_receptor_map()

Build the map between VCC-id and Receptor-id and viceversa. The funciton stores them in instance attributes.

_get_vcc_to_receptor_map()

Get VCC-id vs Receptor-index map from CBF and convert it to a dictionary.

_get_receptor_to_vcc_map()

Get Receptor-index vs VCC-id map from CBF and convert it to a dictionary.

connect() → ska_csp_lmc_common.connector.Connector

Establish a connection with the Mid CBF Controller. On successfull connection , the VCCs/receptors mapping is retrieved.

Returns
An instance of the Connector class on success, otherwise None.

Mid.PSS Controller Component

class ska_csp_lmc_mid.controller.mid_ctrl_component.**MidPssControllerComponent**(**args: Any*, ***kwargs: Any*)

Bases: PssControllerComponent

Specialization of the PssController component class for the Mid.CSP.

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

Initialize the MidPssControllerComponent.

Parameters

- **fqdn** – The Mid.CBF controller TANGO Device Fully Qualified Domain Name.
- **logger** – The device or python logger if default is None.

property list_of_beams

Return the list of PSS beams IDs.

Mid.CBF Subarray Component

class ska_csp_lmc_mid.subarray.mid_cbf_subarray_component.**MidCbfSubarrayComponent**(**args: Any*, ***kwargs: Any*)

Bases: CbfSubarrayComponent

Specialization of the CbfSubarray component class for the Mid.CSP.

property assigned_receptors: List[str]

Return the information about the receptors assigned to the subarray.

Returns

The list of receptors currently assigned to Mid Cbf subarray on success, otherwise an empty list.

_assign_receptors_validator(*receptor_list: Any*) → List[str]

Helper method to validate the list of receptors received as argument of the *AssignResources command*, filtering out the receptors IDs that cannot be assigned to the subarray because already assigned or not available.

The method gets the input list and checks the required receptors IDs against the list of available receptors. This list includes the receptors that:

- have a valid network link with the corresponding VCC
- the connected VCC is enabled and working.
- is not assigned to another subarray.

Parameters

receptor_list – list of receptors IDs to be assigned to the Mid.CBF subarray, as specified by the input argument of the *AssignResources* method.

Returns

the list of valid receptors IDs to assign to the Mid.CBF Subarray on success, otherwise an empty list.

_get_csp_ctrl_connector()

Method that create the CSP controller proxy if not already defined and return it.

_receptor_id_available(*receptor_id: str*, *check_activated: Optional[bool] = True*) → str

Method that performs some the folliwing check:

- the receptor id is stored in the available lists
- the receptor is already assigned to a subarray

Parameters

- **receptor_id** – Ascii receptor id (i.e. SKA001, MKT000)
- **check_activated** – optional flag. If True additional checks are performed on the receptor_id

Returns

valid ascii receptor id if valid. Else empty string.

_receptor_id_validation(*receptor_id: str*) → bool

Verify if receptor name is compliant with ADR32 Dish type: SKA dish [SKA] and MeerKAT dish [MKT]. MeerKAT range 0 to 63 SKA range 1 to 133 i.e: MKT063, SKA001

Parameters

receptor_id – Ascii receptor id (i.e SKA001, MKT000)

Returns

True if receptor name is compliant to ADR32

_remove_receptors_validator(*receptor_list: List[str]*) → List[str]

Method to filter out the receptors IDs that do not belong to the subarray. Validate the list of receptors received as argument of the.

RemoveResources command, filtering out the receptors IDs that do not belong to the subarray.

Parameters

receptor_list – list of receptors IDs to remove from the CBF subarray.

Returns

the list of valid receptors IDs to remove from the MID CBF Subarray on success, otherwise an empty list.

_validated_receptors(*resources: dict*, *validator_function: Callable*) → List[str]

Return the list of the validated receptors IDs that are passed as input argument to the Mid.CBF subarray Add/RemoveReceptors command.

Parameters

- **resources** – the input dictionary with the list of receptors to add/remove to/from the subarray.
- **validator_function** – The callback invoked to validate the list of receptors.

Returns

the list with the receptors to add/remove.

releaseresources(*input_resources: Dict*, *callback: Callable*) → None

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

Parameters

- **input_resources** – The release resource dictionary with configuration

- **callback** – Method invoked when the command ends on the target device

Raise
ValueError exception if the list of receptors is empty.

assignresources(*input_resources: Dict*, *callback: Callable*) → None

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

Parameters

- **input_resources** – The assign resource dictionary with configuration
- **callback** – Method invoked when the commands end on the target device

Raise
ValueError exception if the list of receptors is empty.

assignresources_succeeded()

Succeeded callback invoked on the component

Returns
the list of assigned receptors

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

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

Parameters
callback – Method invoked when the commands end on the target device

Returns
None

Raise
ValueError exception if the list of receptors specified into the configuration dictionary is not valid

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,

Mid.PSS Subarray Component

class ska_csp_lmc_mid.subarray.mid_pss_subarray_component.**MidPssSubarrayComponent**(**args: Any*, ***kwargs: Any*)

Bases: PssSubarrayComponent

Specialization of the PssSubarray component class for the Mid.CSP.

releaseresources(*input_resources: Any*, *callback: Callable*) → None

Invoke the release resources command on the Mid CBF subarray.

Parameters

- **input_resources** – The release resource dictionary with configuration

- **callback** – Method invoked when the commands end on the target device

Returns
None

Raise
ValueError exception if the list of receptors specified into the configuration dictionary is not valid

assignresources(*input_resources: Any*, *callback: Callable*) → None

Invoke the assign resources command on the Mid CBF subarray.

Parameters

- **input_resources** – The assign resource dictionary with configuration
- **callback** – Method invoked when the commands end on the target device

Returns
None

Raise
ValueError exception if the list of receptors specified into the configuration dictionary is not valid

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

Invoke the releaseall resources command on the Mid CBF subarray.

Parameters
callback – Method invoked when the commands end on the target device

Returns
None

Raise
ValueError exception if the list of receptors specified into the configuration dictionary is not valid

CHAPTER

FOUR

MID CSP TANGO CLIENTS EXAMPLES

In the following sections some itango interface examples are provided, to be easily applied to a generic python client.

Basic assumptions for each example are:

1. the system has been fresh initialized
2. the only CSP sub-system deployed is the CBF
3. CBF supports up to 4 VCCs and 4 FSPs
4. All TANGO operations (read/write/command_inout) are always successfully. No check on the results is done in the following examples.

To control CSP with itango, a proxy to CSP Controller and Subarrays has to be created:

```
csp_ctrl = tango.DeviceProxy('mid-csp/control/0')
csp_sub1 = tango.DeviceProxy('mid-csp/subarray/01')
csp_sub2 = tango.DeviceProxy('mid-csp/subarray/02')
csp_sub3 = tango.DeviceProxy('mid-csp/subarray/03')
```

It is possible also to create proxies to CBF controller and subarrays, in order to check their states and mode after a command is issued:

```
cbf_ctrl = tango.DeviceProxy('mid_csp_cbf/sub_elt/controller')
cbf_sub1 = tango.DeviceProxy('mid_csp_cbf/sub_elt/subarray_01')
cbf_sub2 = tango.DeviceProxy('mid_csp_cbf/sub_elt/subarray_02')
cbf_sub3 = tango.DeviceProxy('mid_csp_cbf/sub_elt/subarray_03')
```

Please note that all commands on subarray follow the ObsState state model defined in ADR-8. A CSP.LMC Subarray doesn't allow commands from observing states other than those specified in this model.

After the deployment, all the Mid CSP.LMC devices are in following conditions: - state DISABLE. - healthState UNKNOWN - adminMode OFFLINE - obstState EMPTY (subarrays)

From now on, all the examples refers to these proxy objects. For Subarray commands, the proxy to subarray 1 will be used.

4.1 Mid CSP.LMC start communication

The Mid CSP.LMC Devices *adminMode* is a memorized attribute. That means that it is stored in the TANGO DB and its value is written to the devices just after the initialization. If the *adminMode* is MAINTENANCE(2)/ONLINE(3) the connection between Mid CSP.LMC Devices and the subsystem starts immediately. Otherwise, to start the communication use the following command:

```
csp_ctrl.adminMode = 2  #set to MAINTENANCE
```

or

::

    csp_ctrl.adminMode = 3 #set to ONLINE

Mid CSP.LMC Controller forwards the *adminMode* value to its Subarrays and subordinated systems devices. The new states are the following for all the devices:

```
- state OFF
- healthState OK
- adminMode MAINTENANCE(2)/ONLINE(3)
- obstState EMPTY (subarrays)
```

4.2 Power-on (off/standby) the Mid.CSP

To power on the Mid.CSP devices, issue the command:

```
csp_ctrl.On([]) # empty list: power on all the available sub-systems (CBF, PSS, PST)
```

or

```
csp_ctrl.On(['mid_csp_cbf/sub_elt/controller', ]) # power-on only the specified sub-
↪systems
```

The command returns immediately the following list:

::

    [array([2], dtype=int32), ['1679401117.9451234_224758016395799_On']]

Where:

- 2 is the command status (2 means QUEUED)
- '1679401117.9451234_224758016395799_On' is a unique id assigned to the command. It can be used to track the execution status of the command

It is possible to read the command result state using:

```
cmd_result = csp_ctrl.commandResult
```

cmd_result is a Tuple of two strings:

- the first element is the name of last executed CSP task
- the second one is the result code: allowed values for the result code are defined in SKA Base Classes module ska_tango_base.commands

Possible results for the current example are:

- (‘on’, ‘0’) # *On* task completed successfully
- (‘on’, ‘1’) # *On* task started
- (‘on’, ‘3’) # *On* task completed with failure

Some of the long running command attributes can also be accessed. Long running command result can be read using:

```
long_running_command_result = csp_ctrl.longRunningCommandResult
```

long_running_command_result is a Tuple of a string and a list:

- the first element is a command id assigned when command is invoked
- the second element is a list of result code (matching the value in command result attribute described above) and result message

Possible results:

- (‘1684312814.139426_265125881596693_On’, ‘[0, “on completed 1/1”]’) # *On* task completed successfully on one devices
- (‘1684312814.139426_265125881596693_Configure’, ‘[0, “configure completed on components 2/2”]’) # *Configure* task completed successfully on two devices
- (‘1684312814.139426_265125881596693_Off’, ‘[3, “off completed 1/1”]’) # *Off* task completed with failure on one device

It is also possible to access long running command status at the various stages of command execution. This attribute can be read using:

```
long_running_command_status = csp_ctrl.longRunningCommandStatus
```

long_running_command_status is a Tuple of pairs of strings. In each pair the elements represent the following:

- the first element is a command id assigned when command is invoked
- the second element is the task status: allowed values for the task status are defined in SKA Base Classes module ska_tango_base.executor

Possible results:

- (‘1684312814.139426_265125881596693_On’, ‘QUEUED’)
- (‘1684312814.139426_265125881596693_On’, ‘IN_PROGRESS’)
- (‘1684312814.139426_265125881596693_On’, ‘COMPLETED’)
- (‘1684312814.139426_265125881596693_On’, ‘ABORTED’)
- (‘1684312814.139426_265125881596693_On’, ‘FAILED’)
- (‘1684312814.139426_265125881596693_On’, ‘REJECTED’)

Note that there can be more than one pair of command id and task status in the attribute, if there are more commands invoked.

The command *On* invoked on the Mid.CSP Controller is forwarded to the Mid CBF sub-system Controller and to all the Mid.CSP Subarrays. This can be checked by the state of all controllers and subarrays:

```
csp_ctrl.state() -> ON
csp_sub1.state() -> ON
csp_sub2.state() -> ON
csp_sub3.state() -> ON
```

(continues on next page)

(continued from previous page)

```
cbf_ctrl.state() -> ON
cbf_sub1.state() -> ON
cbf_sub2.state() -> ON
cbf_sub3.state() -> ON
```

The same logic and syntax apply also for *Off* and *Standby* commands.

4.3 Assign resources to a Mid CSP.LMC Subarray

To assign resources to a subarray both the subarray and controller devices must be in ON operational state. To move the system in such a state, please follow the previous example. Please note that right now only the receptor ids SKA001, SKA022, SKA103, SKA104 can be assigned to a subarray.

The following JSON string can be used to assign receptors to the Mid CSP.LMC:

```
json_string = '{"subarray_id": 1,"dish":{ "receptor_ids":["SKA001", "SKA022"]}}'
```

On successfully assignment, the receptors SKA001 and SKA022 are affiliated to subarray 1.

Invoke the *AssignResources* command on Mid.CSP Subarray 1:

```
csp_sub1.assignresources(json_string)
```

If command is successful, the command result will report:

```
csp_sub1.commandResult -> ('assignresources', '0')

csp_sub1.commandResultName -> 'assignresources'
csp_sub1.commandResultCode -> '0'
```

The receptors assigned to Mid.CSP Subarray 1 are:

```
csp_sub1.assignedReceptors -> ("SKA001", "SKA022")
```

Information about resources availability are provided by the Mid.CSP Controller.

To get the list of all the receptors:

```
csp_ctrl.receptorsList -> ("SKA001", "SKA022", "SKA103", "SKA104")
```

To get the list of the available receptors:

```
csp_ctrl.unassignedReceptorIDs -> ("SKA103", "SKA104")
```

To get the affiliation of the receptors to the subarrays:

```
csp_ctrl.receptorMembership -> [1, 1, 0, 0]
```

After resource allocation the Mid CSP.LMC and Mid CBF Subarray *obsState* attribute value changes from EMPTY to IDLE. To check the observing state of the devices:

```
csp_sub1.obsstate -> IDLE
cbf_sub1.obsstate -> IDLE
```

4.4 Configure, issue and end a scan

After a Subarray has resources assigned, it is possible to configure it and then start a scan.

The `json_string` to be used for configure and scan can be found *here*. They have to be assigned to a variable and sent as command input as showed above for assignresources.

First of all, Configure command has to be issued:

```
csp_sub1.configure(json_string_configure)
```

The observimg state will be in CONFIGURING during the execution. After that, if the command is successful:

```
csp_sub1.commandResult -> ('configure', '0')
csp_sub1.obsstate -> READY
cbf_sub1.obsstate -> READY
```

The subarray in READY observing state can be re-configured with a new configuration that overwrites the previous one.

When the subarray is READY, a scan can be started, issuing the *Scan* command:

```
csp_sub1.scan(json_string_scan)
```

If the command is successful:

```
csp_sub1.commandResult -> ('scan', '1')
csp_sub1.obsstate -> SCANNING
cbf_sub1.obsstate -> SCANNING
```

Note that the result code associated to the *Scan* command will remain '1' for all the duration of the scanning process. In fact, according to ADR-8 a scan can be interrupted by the *EndScan* or the *Abort* command. The *Abort* command has to be intended as an emergency call that interrupts abruptly the scan process. On the other side, the *EndScan* first ensures that all the processes are correctly managed.

To end a scan, just issue:

```
csp_sub1.endscan()
```

After *EndScan* is successful, the subarray *obsState* is READY, and another scan can be issued with the same configuration.

On the other side, if the scan is aborted, the *obsState* will go (after a short time in ABORTING) to ABORTED state To perform a new scanning, the subarray observation should be restarted (via the *ObsReset* command) and a new configuration need to be sent (ADR-8)

The sequence of operation is:

```
csp_sub1.abort()
csp_sub1.commandResult -> ('abort', '1')
csp_sub1.obsstate -> ABORTING
csp_sub1.commandResult -> ('abort', '0')
csp_sub1.obsstate -> ABORTED
csp_sub1.obsreset()
csp_sub1.commandResult -> ('obsreset', '1')
csp_sub1.obsstate -> RESETTING
```

(continues on next page)

(continued from previous page)

```
csp_sub1.commandResult -> ('obsreset', '0')
csp_sub1.obsstate -> IDLE
```

4.5 Go To Idle and Release Resources

The resources of a subarray can only be released when its obsState is IDLE. When the subarray is in READY (as happens after the end of a scan) it must first be sent to IDLE with the command:

```
csp_sub1.gotoidle()
```

Upon successful completion of the command, the *obsState* will be IDLE and the resources can be partially or totally removed from the subarray.

To partially remove some of the allocated resources, a json string, like the one used for assign resources (see above) should be sent. This string must specify the receptors to be removed.

Following the previous example, to remove the receptor 1 from the list of the assigned receptors to the Mid CSP.LMC Subarray 1:

```
json_string = '{"subarray_id": 1,"dish":{ "receptor_ids":["SKA001"]}}
csp_sub1.ReleaseResources(json_string)
```

On command success, the subarray will have only receptor 2 assigned to it, and its *obsState* will stay in IDLE. The released receptor (1) will now appear in the pool of the Mid CSP.LMC available resources. This can be verified, accessing the proper attributes of the Mid CSP.LMC Subarray and Controller devices:

```
csp_sub1.commandResult -> ('releaseresources', '0')
csp_sub1.obsstate -> IDLE

csp_sub1.assignedReceptors -> ["SKA022"]
csp_ctrl.receptorsList -> ["SKA001", "SKA022", "SKA103", "SKA104"]
csp_ctrl.unassignedReceptorIDs -> ["SKA001", "SKA103", "SKA104"]
csp_ctrl.receptorMembership -> [0, 1, 0, 0]
```

Otherwise, if all resources are meant to be removed, this can be done with the *ReleaseAllResouces* command:

```
csp_sub1.ReleaseAllResources()
```

On command success, the subarray will be EMPTY again:

```
csp_sub1.commandResult -> ('releaseallresources', '0')
csp_sub1.obsstate -> EMPTY

csp_sub1.assignedReceptors -> []
csp_ctrl.unassignedReceptorIDs -> ["SKA001", "SKA022", "SKA103", "SKA104"]
csp_ctrl.receptorMembership -> [0, 0, 0, 0]
```

4.6 Recover a MID CSP>LMC observing TANGO Device from a FAULT obsState: *Restart*

If something goes wrong with the observation, the *obsState* of the Mid CSP.LMC Subarray could transition to FAULT. Note that this condition differs from the one where the State of a TANGO Device is in FAULT (see below). To recover from this situation, issue the *Restart* command. This command will release also all the resources, bringing the subarray into an EMPTY *obsState*.

The sequence of operations and responses is:

```
csp_sub1.obsState -> FAULT

csp_sub1.Restart()

csp_sub1.commandResult -> ('restart', '1')
csp_sub1.obsState -> RESTARTING
csp_sub1.commandResult -> ('restart', '0')
csp_sub1.obsState -> EMPTY
```

4.7 Recover a Mid CSP.LMC TANGO Device from a FAULT State: *Reset*

When an internal error occours in the Mid CSP.LMC Subarray or in the Controller, the operational state (State) of the device can transition to FAULT. To recover from this condition, the *Reset* command needs to be issued. Successufull execution of this command will return the device in its initial state, that is OFF/EMPTY for Subarray and STANDBY for the Controller.

4.8 Turning OFF a Subarray

The *Off* command disables any signal processing capability of a subarray and all its allocated resources are also released. As for the ADR-8, this command can be issued fron *any* observing state.

Depending on the current observing state of the Mid CSP.LMC Subarray, the *Off* command can be replaced by a sequence of commands that properly bring the device in the desired final state. An approach that works for nearly all the observing states is the following one, where the *Off* command is replaced by the following commands, executed one after the other: - *Abort*: transition the subarray from the current observing state to ABORTED. This command can be issued from all the observing states except: EMPTY and FAULT. In these cases, this step is skipped and the first command invoked must be *Restart*. - Restart: transition the subarray from ABORTED to EMPTY/ON - Off: transition the subarray from ON to OFF.

For further details of the intermediate commands, see examples above.

4.9 Connect to the FSP Processing Modes Capability Device

To connect to the capability device::
fsp_cap_proxy = tango.DeviceProxy('mid-csp/capability-fsp/0') fsp_cap_proxy.state() = DevState.ON

After Controller power-up:

```
fsp_cap_proxy.fspAvailable = [1,2,3,4]
```

After configuring Mid CSP Subarray 1 for correlation using FSP1 and FSP2::
fsp_cap.fspCorrelation = [1] fsp_cap.fspFunctionMode = ['CORR', 'CORR', 'IDLE', 'IDLE'] fsp_cap.Available = 2

CHAPTER

FIVE

JSON COMMAND INPUT TEMPLATES

The following templates can be used as input for the specific command. They are those used in Mid CSP.LMC integration tests.

5.1 Assign Resources

```
{
    "subarray_id": 1,
    "dish":{
        "receptor_ids":["SKA001", "SKA022", "SKA103", "SKA104"]
    },
    "pss":{
        "beams_id":[1,2,3]
    },
    "pst":{
        "beams_id":[1, 2]
    }
}
```

5.2 Configure

```
{
    "interface": "https://schema.skao.int/ska-csp-configure/2.0",
    "subarray": {
        "subarray_name": "science period 23"
    },
    "common": {
        "config_id": "sbi-mvp01-20200325-00001-science_A",
        "frequency_band": "1",
        "subarray_id": 1
    },
    "cbf": {
        "delay_model_subscription_point": "ska_mid/tm_leaf_node/csp_subarray_01/
↪delayModel",
        "fsp": [
            {
                "fsp_id": 1,
```

(continues on next page)

(continued from previous page)

```
                "function_mode": "CORR",
                "frequency_slice_id": 1,
                "integration_factor": 1,
                "zoom_factor": 0,
                "channel_averaging_map": [
                    [0, 2],
                    [744, 0]
                ],
                "channel_offset": 0,
                "output_link_map": [
                    [0, 0],
                    [200, 1]
                ]
            },
            {
                "fsp_id": 2,
             "function_mode": "CORR",
                "frequency_slice_id": 2,
                "integration_factor": 1,
                "zoom_factor": 1,
                "zoom_window_tuning": 650000,
                "channel_averaging_map": [
                    [0, 2],
                    [744, 0]
                ],
                "channel_offset": 744,
                "output_link_map": [
                    [0, 4],
                    [200, 5]
                ],
                "output_host": [
                    [0, "192.168.1.1"]
                ],
                "output_port": [
                    [0, 9744, 1]
                ]
            }
        ],
    "vlbi": {}
    },
    "pss": {
            "pss_beams": [
                    {
                        "pss_beam": 1
                    },
                    {
                        "pss_beam": 2
                    }
            ]
    },
    "pst": {},
    "pointing": {
```

(continues on next page)

(continued from previous page)

```
    "target": {
        "system": "ICRS",
        "target_name": "Polaris Australis",
        "ra": "21:08:47.92",
        "dec": "-88:57:22.9"
    }
    }
}
```

5.3 Scan

```
{
    "interface": "https://schema.skao.int/ska-csp-scan/2.2",
    "scan_id": 11
}
```

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

INDEX

Symbols

M

S