developer.skatelescope.org Documentation

Release 0.1.3

Marco Bartolini

Feb 24, 2022

API

1 API 3

2 Indices and tables 71

Python Module Index 73

Index 75

This project is developing the Local Monitoring and Control (LMC) prototype for the Square Kilometre Array.

CHAPTER

ONE

API

1.1 Controller subpackage

This subpackage implements sat controller functionality for SAT.LMC.

class ControllerComponentManager(*maser_fqdns*, *logger*, *communication_status_changed_callback*, *maser_health_changed_callback*)

A component manager for an SAT LMC controller.

__init__(*maser_fqdns*, *logger*, *communication_status_changed_callback*, *maser_health_changed_callback*)

Initialise a new instance.

Parameters

- **maser_fqdns** (typing.Iterable[str]) – FQDNS of all maser devices
- **logger** (logging.Logger) – the logger to be used by this object.
- **communication_status_changed_callback** (typing.Callable[[*ska_sat_lmc.component.component_manager.CommunicationStatus*], None]) – callback to be called when the status of the communications channel between the component manager and its component changes
- **maser_health_changed_callback** (typing.Callable[[str, typing.Optional[ska_tango_base.control_model.HealthState]], None]) – callback to be called when the health of the maser changes

off()

Turn off the SatLMC subsystem.

Return type ska_tango_base.commands.ResultCode

Returns a result code

on()

Turn on the SatLMC subsystem.

Return type ska_tango_base.commands.ResultCode

Returns a result code

start_communicating()

Establish communication with its components.

Return type None

stop_communicating()

Break off communication with its components.

Return type None

class ControllerHealthModel(*maser_fqdns*, *health_changed_callback*)
A health model for a controller.

__init__(*maser_fqdns*, *health_changed_callback*)
Initialise a new instance.

Parameters

- **maser_fqdns** (typing.Sequence[str]) – the FQDNs of the masers
- **health_changed_callback** (typing.Callable[[ska_tango_base.control_model.HealthState], None]) – callback to be called whenever there is a change to this health model's evaluated health state.

evaluate_health()
Compute overall health of the controller.

The overall health is based on the fault and communication status of the controller overall, together with the health of the subservient devices that it manages.

This implementation simply sets the health of the controller to the health of its least healthy component.

Return type ska_tango_base.control_model.HealthState

Returns an overall health of the controller

maser_health_changed(*maser_fqdn*, *maser_health*)
Handle a change in maser health.

Parameters

- **maser_fqdn** (str) – the FQDN of the maser whose health has changed
- **maser_health** (typing.Optional[ska_tango_base.control_model.HealthState]) – the health state of the specified maser, or None if the maser's admin mode indicates that its health should not be rolled up.

Return type None

class SatController(**args*, ***kwargs*)
An implementation of a controller Tango device for SatLMC.

class InitCommand(**args*, ***kwargs*)
A class for *SatController*'s Init command.

The *do()* method below is called during *SatController*'s initialisation.

do()
Initialise the attributes and properties.

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

create_component_manager()
Create and return a component manager for this device.

Return type *ska_sat_lmc.controller.controller_component_manager.ControllerComponentManager*

Returns a component manager for this device.

health_changed(*health*)
Call this method whenever the HealthModel's health state changes.

Responsible for updating the tango side of things i.e. making sure the attribute is up to date, and events are pushed.

Parameters **health** (ska_tango_base.control_model.HealthState) – the new health value

Return type None

init_command_objects()
Set up the handler objects for Commands.

Return type None

init_device()
Initialise the device.

This is overridden here to change the Tango serialisation model.

Return type None

1.1.1 Controller Component Manager

This module implements component management for the SAT.LMC controller.

class ControllerComponentManager(*maser_fqdns*, *logger*, *communication_status_changed_callback*, *maser_health_changed_callback*)
A component manager for an SAT LMC controller.

__init__(*maser_fqdns*, *logger*, *communication_status_changed_callback*, *maser_health_changed_callback*)
Initialise a new instance.

Parameters

- **maser_fqdns** (typing.Iterable[str]) – FQDNS of all maser devices
- **logger** (logging.Logger) – the logger to be used by this object.
- **communication_status_changed_callback** (typing.Callable[[*ska_sat_lmc.component.component_manager.CommunicationStatus*], None]) – callback to be called when the status of the communications channel between the component manager and its component changes
- **maser_health_changed_callback** (typing.Callable[[str, typing.Optional[ska_tango_base.control_model.HealthState]], None]) – callback to be called when the health of the maser changes

off()
Turn off the SatLMC subsystem.

Return type ska_tango_base.commands.ResultCode

Returns a result code

on()
Turn on the SatLMC subsystem.

Return type ska_tango_base.commands.ResultCode

Returns a result code

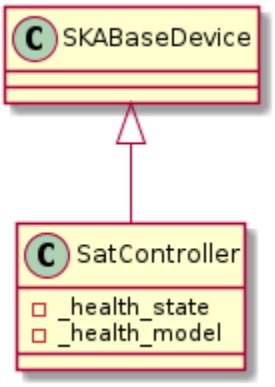
start_communicating()
Establish communication with its components.

Return type None

stop_communicating()
Break off communication with its components.

Return type None

1.1.2 Controller Device



This module contains the ska_sat_lmc Controller device prototype.

class SatController(**args*, ***kwargs*)

An implementation of a controller Tango device for SatLMC.

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

A class for *SatController*'s Init command.

The *do()* method below is called during *SatController*'s initialisation.

do()

Initialise the attributes and properties.

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

create_component_manager()

Create and return a component manager for this device.

Return type *ska_sat_lmc.controller.controller_component_manager.ControllerComponentManager*

Returns a component manager for this device.

health_changed(*health*)

Call this method whenever the HealthModel's health state changes.

Responsible for updating the tango side of things i.e. making sure the attribute is up to date, and events are pushed.

Parameters **health** (ska_tango_base.control_model.HealthState) – the new health value

Return type None

init_command_objects()

Set up the handler objects for Commands.

Return type None

init_device()

Initialise the device.

This is overridden here to change the Tango serialisation model.

Return type None

main(**args*, ***kwargs*)
Entry point for module.

Parameters

- **args** (str) – positional arguments
- **kwargs** (str) – named arguments

Return type int

Returns exit code

1.1.3 Controller Health Model

An implementation of a health model for a controller.

class ControllerHealthModel(*maser_fqdns*, *health_changed_callback*)
A health model for a controller.

__init__(*maser_fqdns*, *health_changed_callback*)
Initialise a new instance.

Parameters

- **maser_fqdns** (typing.Sequence[str]) – the FQDNs of the masers
- **health_changed_callback** (typing.Callable[[ska_tango_base.control_model.HealthState], None]) – callback to be called whenever there is a change to this health model's evaluated health state.

evaluate_health()
Compute overall health of the controller.

The overall health is based on the fault and communication status of the controller overall, together with the health of the subservient devices that it manages.

This implementation simply sets the health of the controller to the health of its least healthy component.

Return type ska_tango_base.control_model.HealthState

Returns an overall health of the controller

maser_health_changed(*maser_fqdn*, *maser_health*)
Handle a change in maser health.

Parameters

- **maser_fqdn** (str) – the FQDN of the maser whose health has changed
- **maser_health** (typing.Optional[ska_tango_base.control_model.HealthState]) – the health state of the specified maser, or None if the maser's admin mode indicates that its health should not be rolled up.

Return type None

1.2 Maser subpackage

This subpackage implements maser functionality for SAT.LMC.

class MaserComponentManager(*initial_simulation_mode*, *initial_test_mode*, *logger*, *maser_url*, *simulator_url*, *communication_status_changed_callback*, *component_fault_callback*)

A component manager that handles a Maser simulator and driver.

__init__(*initial_simulation_mode*, *initial_test_mode*, *logger*, *maser_url*, *simulator_url*, *communication_status_changed_callback*, *component_fault_callback*)

Initialise a new instance.

Parameters

- **initial_simulation_mode** (ska_tango_base.control_model.SimulationMode) – the simulation mode that the component should start in
- **initial_test_mode** (ska_tango_base.control_model.TestMode) – the simulation mode that the component should start in
- **logger** (logging.Logger) – a logger for this object to use
- **maser_url** (str) – the URL of the maser
- **simulator_url** (str) – the URL of the simulated maser
- **communication_status_changed_callback** (typing.Callable[[*ska_sat_lmc.component.component_manager.CommunicationStatus*], None]) – callback to be called when the status of the communications channel between the component manager and its component changes
- **component_fault_callback** (typing.Callable[[bool], None]) – callback to be called when the component faults (or stops faulting)

property simulation_mode: ska_tango_base.control_model.SimulationMode

Return the simulation mode.

Return type ska_tango_base.control_model.SimulationMode

Returns the simulation mode

property test_mode: ska_tango_base.control_model.TestMode

Return the test mode.

Return type ska_tango_base.control_model.TestMode

Returns the test mode

class MaserDriver(*initial_simulation__mode*, *initial_test_mode*, *logger*, *maser_url*, *simulator_url*, *communication_status_changed_callback*, *component_fault_callback*)

Device Server for the T4Science i3000 Maser for SAT.LMC.

__init__(*initial_simulation__mode*, *initial_test_mode*, *logger*, *maser_url*, *simulator_url*, *communication_status_changed_callback*, *component_fault_callback*)

Initialise the attributes and properties of the SatMaser.

Parameters

- **initial_simulation__mode** (ska_tango_base.control_model.SimulationMode) – the simulation mode that the component should start in
- **initial_test_mode** (ska_tango_base.control_model.TestMode) – the simulation mode that the component should start in

- **logger** (logging.Logger) – a logger for this object to use
- **maser_url** (str) – the URL of the hardware maser
- **simulator_url** (str) – the URL of the simulated maser
- **communication_status_changed_callback** (typing.Callable[[*ska_sat_lmc.component.component_manager.CommunicationStatus*], None]) – callback to be called when the status of the communications channel between the component manager and its component changes
- **component_fault_callback** (typing.Callable[[bool], None]) – callback to be called when the component faults (or stops faulting)

property ambient_temperature: float
Report the ambient temperature.

Return type float

Returns ambient temperature

property amplitude_405khz_voltage: float
Report the 405 kHz Amplitude.

Return type float

Returns the 405 kHz Amplitude

property battery_current_a: float
Report the current of battery A.

Return type float

Returns battery current A

property battery_current_b: float
Report the current of battery B.

Return type float

Returns battery current B

property battery_voltage_a: float
Report the voltage of battery A.

Return type float

Returns battery voltage A

property battery_voltage_b: float
Report the voltage of battery B.

Return type float

Returns battery voltage B

property boxes_current: float
Report the boxes current.

Return type float

Returns boxes current

property boxes_temperature: float
Report the boxes temperature.

Return type float

Returns boxes temperature

property cfield_voltage: float
Report the C-field voltage.

Return type float

Returns C-field voltage

property dissociator_current: float
Report the dissociator current.

Return type float

Returns dissociator current

property dissociator_light: float
Report the dissociator light.

Return type float

Returns dissociator light

property external_bottom_heater_voltage: float
Report the external bottom heater voltage.

Return type float

Returns external bottom heater voltage

property external_high_current_value: float
Report the external high current value.

Return type float

Returns external high current value

property external_high_voltage_value: float
Report the external high voltage value.

Return type float

Returns external high voltage value

property external_side_heater_voltage: float
Report the external side heater voltage.

Return type float

Returns external side heater voltage

property hydrogen_pressure_measured: float
Report the measured hydrogen pressure.

Return type float

Returns hydrogen pressure measurement

property hydrogen_pressure_setting: float
Report the hydrogen pressure setting.

Return type float

Returns hydrogen pressure setting

property hydrogen_storage_heater_voltage: float
Report the hydrogen storage heater voltage.

Return type float

Returns hydrogen storage heater voltage

property hydrogen_storage_pressure: float
Report the hydrogen storage pressure.

Return type float

Returns hydrogen storage pressure

property internal_bottom_heater_voltage: float
Report the internal bottom heater voltage.

Return type float

Returns internal bottom heater voltage

property internal_high_current_value: float
Report the internal high current value.

Return type float

Returns internal high current value

property internal_high_voltage_value: float
Report the internal high voltage value.

Return type float

Returns internal high voltage value

property internal_side_heater_voltage: float
Report the internal side heater voltage.

Return type float

Returns internal side heater voltage

property internal_top_heater_voltage: float
Report the internal top heater voltage.

Return type float

Returns internal top heater voltage

property isolator_heater_voltage: float
Report the isolator heater voltage.

Return type float

Returns isolator heater voltage

property lock100mhz: float
Report the lock 100MHz status.

Return type float

Returns the lock 100MHz status

property negative15vdc: float
Report the -15 V supply voltage.

Return type float

Returns -15 V supply voltage

property negative5vdc: float
Report the -5 V supply voltage.

Return type float

Returns -5 V supply voltage

property oscillator_100mhz_voltage: float
Report the oscillator_voltage 100MHz.

Return type float

Returns oscillator_voltage 100MHz

property oscillator_voltage: float
Report the OCXO varicap voltage.

Return type float

Returns the OCXO varicap voltage

property phase_lock_loop_lockstatus: bool
Report the main PLL lock status.

Return type bool

Returns main PLL lock status

property pirani_heater_voltage: float
Report the pirani heater voltage.

Return type float

Returns pirani heater voltage

property positive15vdc: float
Report the +15 V supply voltage.

Return type float

Returns +15 V supply voltage

property positive18vdc: float
Report the +18 V supply voltage.

Return type float

Returns +18 V supply voltage

property positive24vdc: float
Report the +24 V supply voltage.

Return type float

Returns +24 V supply voltage

property positive5vdc: float
Report the +5 V supply voltage.

Return type float

Returns +5 V supply voltage

property positive8vdc: float
Report the +8 V supply voltage.

Return type float

Returns +8 V supply voltage

property purifier_current: float
Report the purifier current.

Return type float

Returns purifier current

start_communicating()
Establish communication with the maser.

Return type None

stop_communicating()
Stop communicating with the maser.

Return type None

property thermal_control_unit_heater_voltage: float
Report the thermal control unit heater voltage.

Return type float

Returns thermal control unit heater voltage

property tube_heater_voltage: float
Report the tube heater voltage.

Return type float

Returns tube heater voltage

property varactor_diode_voltage: float
Report the varactor voltage.

Return type float

Returns varactor voltage

wait_for_maser_thread_running()
Wait for the maser hardware thread to run.

Return type None

wait_for_simulator_thread_running()
Wait for the simulator thread to run.

Return type None

class MaserHealthModel(*health_changed_callback*)
A health model for a maser.

At present this uses the base health model; this is a placeholder for a future, better implementation.

class SatMaser(args*, ***kwargs*)**
An implementation of a Maser Tango device for SatLmc.

GetVersionInfo()
Get the version the device.

Return type str

Returns Version details of the device.

class InitCommand(args*, ***kwargs*)**
Implement the device initialisation for the Maser device.

do()
Initialise the attributes and properties.
Returns A tuple containing a return code and a string message indicating status. The message is for information purpose only.

Off()
Put the device into off mode.

The Maser is always on so this does nothing

Return type typing.Tuple[typing.List[ska_tango_base.commands.ResultCode], typing.List[typing.Optional[str]]]

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

On()
Put the device into on mode.

The Maser is always on so this does nothing

Return type typing.Tuple[typing.List[ska_tango_base.commands.ResultCode], typing.List[typing.Optional[str]]]

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

Reset()
Reset the device.

The Maser is always on so this does nothing

Return type typing.Tuple[typing.List[ska_tango_base.commands.ResultCode], typing.List[typing.Optional[str]]]

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

Standby()
Put the device into standby mode.

The Maser is always on so this does nothing

Return type typing.Tuple[typing.List[ska_tango_base.commands.ResultCode], typing.List[typing.Optional[str]]]

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

ambient_temperature()
Report the ambient temperature.

Return type float

Returns ambient temperature

amplitude_405khz_voltage()
Report the 405 kHz Amplitude.

Return type float

Returns the 405 kHz Amplitude

battery_current_a()
Report the current of battery A.

Return type float

Returns battery current A

battery_current_b()
Report the current of battery B.

Return type float

Returns battery current B

battery_voltage_a()
Report the voltage of battery A.

Return type float

Returns battery voltage A

battery_voltage_b()
Report the voltage of battery B.

Return type float

Returns battery voltage B

boxes_current()
Report the boxes current.

Return type float

Returns boxes current

boxes_temperature()
Report the boxes temperature.

Return type float

Returns boxes temperature

cfield_voltage()
Report the C-field voltage.

Return type float

Returns C-field voltage

create_component_manager()
Create and return a component manager for this device.

Return type *ska_sat_lmc.maser.maser_driver.MaserDriver*

Returns a component manager for this device.

dissociator_current()
Report the dissociator current.

Return type float

Returns dissociator current

dissociator_light()
Report the dissociator light.

Return type float

Returns dissociator light

external_bottom_heater_voltage()
Report the external bottom heater voltage.

Return type float

Returns external bottom heater voltage

external_high_current_value()
Report the external high current value.

Return type float

Returns external high current value

external_high_voltage_value()
Report the external high voltage value.

Return type float

Returns external high voltage value

external_side_heater_voltage()
Report the external side heater voltage.

Return type float

Returns external side heater voltage

health_changed(*health*)
Handle change in this device's health state.

This is a callback hook, called whenever the HealthModel's evaluated health state changes. It is responsible for updating the tango side of things i.e. making sure the attribute is up to date, and events are pushed.

Parameters **health** (ska_tango_base.control_model.HealthState) – the new health value

Return type None

hydrogen_pressure_measured()
Report the measured hydrogen pressure.

Return type float

Returns hydrogen pressure measurement

hydrogen_pressure_setting()
Report the hydrogen pressure setting.

Return type float

Returns hydrogen pressure setting

hydrogen_storage_heater_voltage()
Report the hydrogen storage heater voltage.

Return type float

Returns hydrogen storage heater voltage

hydrogen_storage_pressure()
Report the hydrogen storage pressure.

Return type float

Returns hydrogen storage pressure

init_command_objects()
Initialise the command handlers for commands supported by this device.

Return type None

init_device()
Initialise the device.

This is overridden here to change the Tango serialisation model.

Return type None

internal_bottom_heater_voltage()
Report the internal bottom heater voltage.

Return type float

Returns internal bottom heater voltage

internal_high_current_value()
Report the internal high current value.

Return type float

Returns internal high current value

internal_high_voltage_value()
Report the internal high voltage value.

Return type float

Returns internal high voltage value

internal_side_heater_voltage()
Report the internal side heater voltage.

Return type float

Returns internal side heater voltage

internal_top_heater_voltage()
Report the internal top heater voltage.

Return type float

Returns internal top heater voltage

is_attribute_allowed(*attr_req_type*)
Protect attribute access before being updated otherwise it reports alarm.

Parameters **attr_req_type** (tango.AttReqType) – tango attribute type READ/WRITE

Return type bool

Returns True if the attribute can be read else False

isolator_heater_voltage()
Report the isolator heater voltage.

Return type float

Returns isolator heater voltage

lock100mhz()
Report the lock 100MHz status.

Return type float

Returns the lock 100MHz status

`negative15vdc()`
Report the -15 V supply voltage.

Return type `float`

Returns -15 V supply voltage

`negative5vdc()`
Report the -5 V supply voltage.

Return type `float`

Returns -5 V supply voltage

`oscillator_100mhz_voltage()`
Report the oscillator_voltage 100MHz.

Return type `float`

Returns oscillator_voltage 100MHz

`oscillator_voltage()`
Report the OCXO varicap voltage.

Return type `float`

Returns the OCXO varicap voltage

`phase_lock_loop_lockstatus()`
Report the main PLL lock status.

Return type `bool`

Returns main PLL lock status

`pirani_heater_voltage()`
Report the pirani heater voltage.

Return type `float`

Returns pirani heater voltage

`positive15vdc()`
Report the +15 V supply voltage.

Return type `float`

Returns +15 V supply voltage

`positive18vdc()`
Report the +18 V supply voltage.

Return type `float`

Returns +18 V supply voltage

`positive24vdc()`
Report the +24 V supply voltage.

Return type `float`

Returns +24 V supply voltage

`positive5vdc()`
Report the +5 V supply voltage.

Return type float

Returns +5 V supply voltage

positive8vdc()
Report the +8 V supply voltage.

Return type float

Returns +8 V supply voltage

purifier_current()
Report the purifier current.

Return type float

Returns purifier current

simulationMode(*value*)
Set the simulation mode.

Parameters **value** (ska_tango_base.control_model.SimulationMode) – The simulation mode, as a SimulationMode value

Return type None

testMode(*value*)
Set the test mode.

Parameters **value** (ska_tango_base.control_model.TestMode) – The test mode, as a TestMode value

Return type None

thermal_control_unit_heater_voltage()
Report the thermal control unit heater voltage.

Return type float

Returns thermal control unit heater voltage

tube_heater_voltage()
Report the tube heater voltage.

Return type float

Returns tube heater voltage

varactor_diode_voltage()
Report the varactor voltage.

Return type float

Returns varactor voltage

1.2.1 Maser Component Manager

This module implements the maser component manager.

class MaserComponentManager(*initial_simulation_mode*, *initial_test_mode*, *logger*, *maser_url*, *simulator_url*, *communication_status_changed_callback*, *component_fault_callback*)

A component manager that handles a Maser simulator and driver.

__init__(*initial_simulation_mode*, *initial_test_mode*, *logger*, *maser_url*, *simulator_url*, *communication_status_changed_callback*, *component_fault_callback*)

Initialise a new instance.

Parameters

- **initial_simulation_mode** (ska_tango_base.control_model.SimulationMode) – the simulation mode that the component should start in
- **initial_test_mode** (ska_tango_base.control_model.TestMode) – the simulation mode that the component should start in
- **logger** (logging.Logger) – a logger for this object to use
- **maser_url** (str) – the URL of the maser
- **simulator_url** (str) – the URL of the simulated maser
- **communication_status_changed_callback** (typing.Callable[[*ska_sat_lmc.component.component_manager.CommunicationStatus*], None]) – callback to be called when the status of the communications channel between the component manager and its component changes
- **component_fault_callback** (typing.Callable[[bool], None]) – callback to be called when the component faults (or stops faulting)

property simulation_mode: ska_tango_base.control_model.SimulationMode

Return the simulation mode.

Return type ska_tango_base.control_model.SimulationMode

Returns the simulation mode

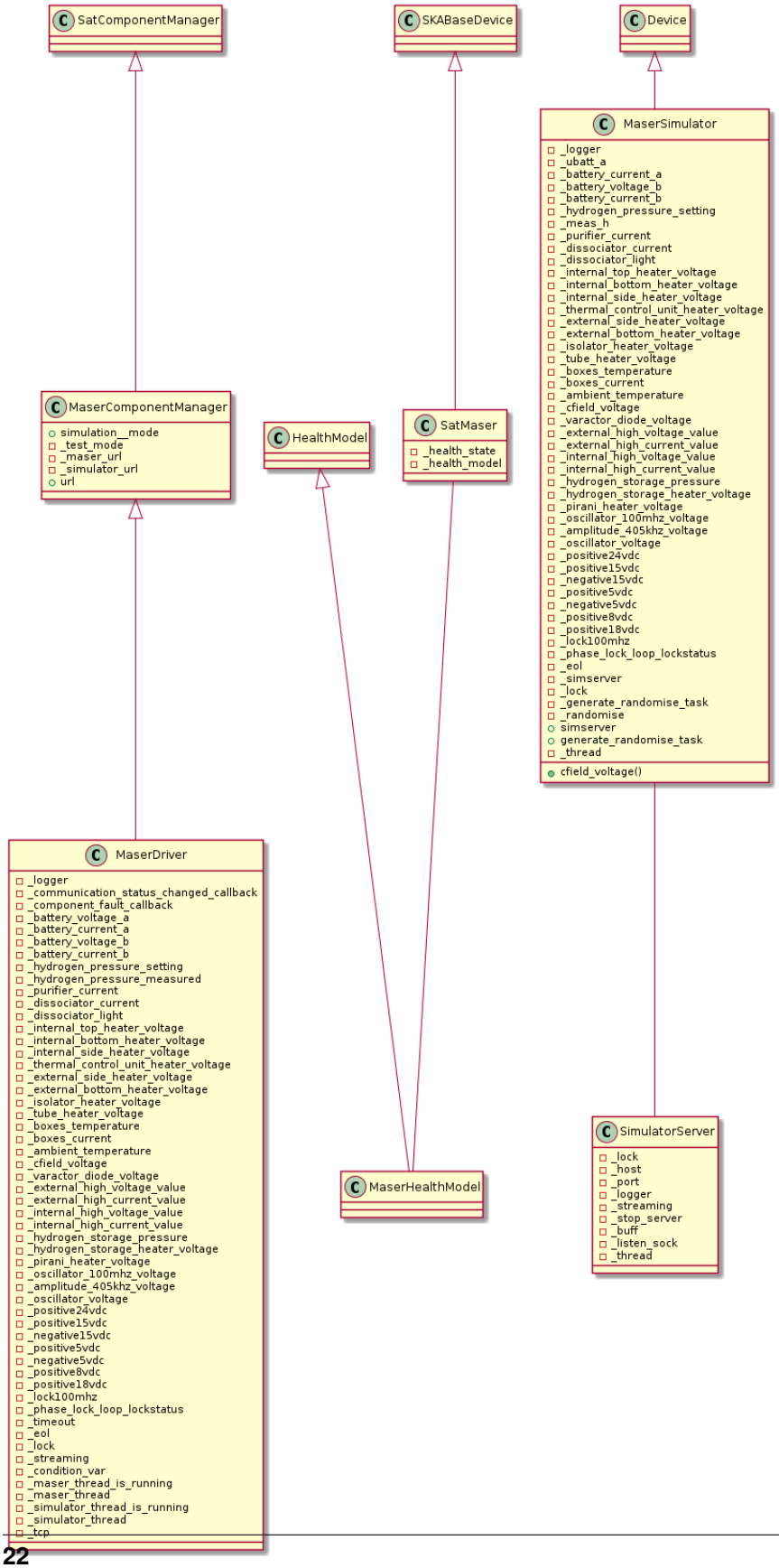
property test_mode: ska_tango_base.control_model.TestMode

Return the test mode.

Return type ska_tango_base.control_model.TestMode

Returns the test mode

1.2.2 Maser Device



This module implements the Sa
Maser device.

class SatMaser(*args, **kwargs)
An implementation of a Maser Tango device for SatLmc.

GetVersionInfo()
Get the version the device.

Return type str

Returns Version details of the device.

class InitCommand(*args, **kwargs)
Implement the device initialisation for the Maser device.

do()
Initialise the attributes and properties.

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

Off()
Put the device into off mode.

The Maser is always on so this does nothing

Return type typing.Tuple[typing.List[ska_tango_base.commands.ResultCode], typing.List[typing.Optional[str]]]

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

On()
Put the device into on mode.

The Maser is always on so this does nothing

Return type typing.Tuple[typing.List[ska_tango_base.commands.ResultCode], typing.List[typing.Optional[str]]]

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

Reset()
Reset the device.

The Maser is always on so this does nothing

Return type typing.Tuple[typing.List[ska_tango_base.commands.ResultCode], typing.List[typing.Optional[str]]]

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

Standby()
Put the device into standby mode.

The Maser is always on so this does nothing

Return type typing.Tuple[typing.List[ska_tango_base.commands.ResultCode], typing.List[typing.Optional[str]]]

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

ambient_temperature()
Report the ambient temperature.

Return type float

Returns ambient temperature

amplitude_405khz_voltage()
Report the 405 kHz Amplitude.

Return type float

Returns the 405 kHz Amplitude

battery_current_a()
Report the current of battery A.

Return type float

Returns battery current A

battery_current_b()
Report the current of battery B.

Return type float

Returns battery current B

battery_voltage_a()
Report the voltage of battery A.

Return type float

Returns battery voltage A

battery_voltage_b()
Report the voltage of battery B.

Return type float

Returns battery voltage B

boxes_current()
Report the boxes current.

Return type float

Returns boxes current

boxes_temperature()
Report the boxes temperature.

Return type float

Returns boxes temperature

cfield_voltage()
Report the C-field voltage.

Return type float

Returns C-field voltage

create_component_manager()
Create and return a component manager for this device.

Return type *ska_sat_lmc.maser.maser_driver.MaserDriver*

Returns a component manager for this device.

dissociator_current()
Report the dissociator current.

Return type float

Returns dissociator current

dissociator_light()
Report the dissociator light.

Return type float

Returns dissociator light

external_bottom_heater_voltage()
Report the external bottom heater voltage.

Return type float

Returns external bottom heater voltage

external_high_current_value()
Report the external high current value.

Return type float

Returns external high current value

external_high_voltage_value()
Report the external high voltage value.

Return type float

Returns external high voltage value

external_side_heater_voltage()
Report the external side heater voltage.

Return type float

Returns external side heater voltage

health_changed(*health*)
Handle change in this device's health state.

This is a callback hook, called whenever the HealthModel's evaluated health state changes. It is responsible for updating the tango side of things i.e. making sure the attribute is up to date, and events are pushed.

Parameters **health** (ska_tango_base.control_model.HealthState) – the new health value

Return type None

hydrogen_pressure_measured()
Report the measured hydrogen pressure.

Return type float

Returns hydrogen pressure measurement

hydrogen_pressure_setting()
Report the hydrogen pressure setting.

Return type float

Returns hydrogen pressure setting

hydrogen_storage_heater_voltage()
Report the hydrogen storage heater voltage.

Return type float

Returns hydrogen storage heater voltage

hydrogen_storage_pressure()
Report the hydrogen storage pressure.

Return type float

Returns hydrogen storage pressure

init_command_objects()
Initialise the command handlers for commands supported by this device.

Return type None

init_device()
Initialise the device.

This is overridden here to change the Tango serialisation model.

Return type None

internal_bottom_heater_voltage()
Report the internal bottom heater voltage.

Return type float

Returns internal bottom heater voltage

internal_high_current_value()
Report the internal high current value.

Return type float

Returns internal high current value

internal_high_voltage_value()
Report the internal high voltage value.

Return type float

Returns internal high voltage value

internal_side_heater_voltage()
Report the internal side heater voltage.

Return type float

Returns internal side heater voltage

internal_top_heater_voltage()
Report the internal top heater voltage.

Return type float

Returns internal top heater voltage

is_attribute_allowed(*attr_req_type*)
Protect attribute access before being updated otherwise it reports alarm.

Parameters **attr_req_type** (tango.AttReqType) – tango attribute type READ/WRITE

Return type bool

Returns True if the attribute can be read else False

`isolator_heater_voltage()`
Report the isolator heater voltage.

Return type `float`

Returns isolator heater voltage

`lock100mhz()`
Report the lock 100MHz status.

Return type `float`

Returns the lock 100MHz status

`negative15vdc()`
Report the -15 V supply voltage.

Return type `float`

Returns -15 V supply voltage

`negative5vdc()`
Report the -5 V supply voltage.

Return type `float`

Returns -5 V supply voltage

`oscillator_100mhz_voltage()`
Report the oscillator_voltage 100MHz.

Return type `float`

Returns oscillator_voltage 100MHz

`oscillator_voltage()`
Report the OCXO varicap voltage.

Return type `float`

Returns the OCXO varicap voltage

`phase_lock_loop_lockstatus()`
Report the main PLL lock status.

Return type `bool`

Returns main PLL lock status

`pirani_heater_voltage()`
Report the pirani heater voltage.

Return type `float`

Returns pirani heater voltage

`positive15vdc()`
Report the +15 V supply voltage.

Return type `float`

Returns +15 V supply voltage

`positive18vdc()`
Report the +18 V supply voltage.

Return type float

Returns +18 V supply voltage

positive24vdc()
Report the +24 V supply voltage.

Return type float

Returns +24 V supply voltage

positive5vdc()
Report the +5 V supply voltage.

Return type float

Returns +5 V supply voltage

positive8vdc()
Report the +8 V supply voltage.

Return type float

Returns +8 V supply voltage

purifier_current()
Report the purifier current.

Return type float

Returns purifier current

simulationMode(*value*)
Set the simulation mode.

Parameters **value** (ska_tango_base.control_model.SimulationMode) – The simulation mode, as a SimulationMode value

Return type None

testMode(*value*)
Set the test mode.

Parameters **value** (ska_tango_base.control_model.TestMode) – The test mode, as a TestMode value

Return type None

thermal_control_unit_heater_voltage()
Report the thermal control unit heater voltage.

Return type float

Returns thermal control unit heater voltage

tube_heater_voltage()
Report the tube heater voltage.

Return type float

Returns tube heater voltage

varactor_diode_voltage()
Report the varactor voltage.

Return type float

Returns varactor voltage

main(**args*, ***kwargs*)
Entry point for module.

Parameters

- **args** (str) – positional arguments
- **kwargs** (str) – named arguments

Return type int

Returns exit code

1.2.3 Maser Health Model

An implementation of a health model for a SatMaser.

class MaserHealthModel(*health_changed_callback*)
A health model for a maser.

At present this uses the base health model; this is a placeholder for a future, better implementation.

1.3 Phase Micro Stepper subpackage

This module contains the phase microstepper functionality.

class PhaseMicroStepperComponentManager(*initial_simulation_mode*, *initial_test_mode*, *logger*, *phase_url*, *simulator_url*, *communication_status_changed_callback*, *component_fault_callback*)
A component manager that handles a PhaseMicroStepper simulator and driver.

__init__(*initial_simulation_mode*, *initial_test_mode*, *logger*, *phase_url*, *simulator_url*, *communication_status_changed_callback*, *component_fault_callback*)
Initialise a new instance.

Parameters

- **initial_simulation_mode** (ska_tango_base.control_model.SimulationMode) – the simulation mode that the component should start in
- **initial_test_mode** (ska_tango_base.control_model.TestMode) – the simulation mode that the component should start in
- **logger** (logging.Logger) – a logger for this object to use
- **phase_url** (str) – the URL of the PhaseMicroStepper
- **simulator_url** (str) – the URL of the simulated PhaseMicroStepper
- **communication_status_changed_callback** (typing.Callable[[*ska_sat_lmc.component.component_manager.CommunicationStatus*], None]) – callback to be called when the status of the communications channel between the component manager and its component changes
- **component_fault_callback** (typing.Callable[[bool], None]) – callback to be called when the component faults (or stops faulting)

property simulation_mode: ska_tango_base.control_model.SimulationMode
Return the simulation mode.

Return type ska_tango_base.control_model.SimulationMode

Returns the simulation mode

property test_mode: ska_tango_base.control_model.TestMode
Return the test mode.

Return type ska_tango_base.control_model.TestMode

Returns the test mode

class PhaseMicroStepperDriver(*initial_simulation__mode*, *initial_test_mode*, *logger*, *phase_url*, *simulator_url*, *communication_status_changed_callback*, *component_fault_callback*)
Device Server for the SAT.LMC PhaseMicrostepper (100MHz).

__init__(*initial_simulation__mode*, *initial_test_mode*, *logger*, *phase_url*, *simulator_url*, *communication_status_changed_callback*, *component_fault_callback*)
Initialise the attributes and properties of the PhaseMicroStepper.

Parameters

- **initial_simulation__mode** (ska_tango_base.control_model.SimulationMode) – the simulation mode that the component should start in
- **initial_test_mode** (ska_tango_base.control_model.TestMode) – the simulation mode that the component should start in
- **logger** (logging.Logger) – a logger for this object to use
- **phase_url** (str) – the URL of the PhaseMicroStepper
- **simulator_url** (str) – the URL of the simulated PhaseMicroStepper
- **communication_status_changed_callback** (typing.Callable[[*ska_sat_lmc.component.component_manager.CommunicationStatus*], None]) – callback to be called when the status of the communications channel between the component manager and its component changes
- **component_fault_callback** (typing.Callable[[bool], None]) – callback to be called when the component faults (or stops faulting)

advance_10megahertz(*advance*)
Advance the output phase of 10MHz OUT signals.

Value in units of 10ns. Acceptable range is from 0 to +9.

Parameters **advance** (int) – advance by this value

Return type str

Returns the command string & value

advance_phase(*advance*)
Advance the output phase of all OUT signals.in units of 1e-15s.

The resolution is about 1e-13. Acceptable range is from -5000000 to +5000000.

Parameters **advance** (int) – advance by this value

Return type str

Returns the command string & value

advance_pps(*advance*)
Advance the phase of PPS OUT signals in units of 10ns.

The acceptable range is limited from -99000000 to 30000000.

Parameters **advance** (int) – advance by this value

Return type str

Returns the command string & value

property first_internal_oscillator: float
Return the status of internal oscillator 1.

value 1 is fixed point 6 decimal digits

Return type float

Returns the status of internal oscillator 1

get_offset_frequency()
Return the offset frequency set in the last setF command.

Return type str

Returns the command and offset frequency in units of 1e-20

property internal_temperature: float
Return the status of internal Celsius temperature.

value 3 is fixed point 2 decimal digits.

Return type float

Returns the internal temperature

phase_thread_running()
Return the state of the PhaseMicroStepper hardware thread.

Return type bool

Returns True if the PhaseMicroStepper thread is running

property second_internal_oscillator: float
Return the status of internal oscillator 2.

value 2 is fixed point 6 decimal digits.

Return type float

Returns the status of internal oscillator 2

property serial_number: int
Return the serial number of the EOG.

Return type int

Returns the serial number

set_ip_address(*ip_address*)
Set the IP address of the EOG and closes the TCP connection.

Parameters **ip_address** (str) – the new IP address

Return type str

Returns the command string & IP address set

set_offset_frequency(*offset*)
Set the offset frequency.

Sets the OUT signals respect to the frequency of 100MHz IN signal, in units of 1e-20. Resolution is better than 3e-20. The acceptable range is from -1000000000000 to 1000000000000.

Parameters **offset** (int) – the offset frequency

Return type str

Returns the command and converted frquency in units of 1e-20

set_udp_address(*udp_address*)
Set the UDP destination address.

If network address is 255.255.255.255, the UDP message is broadcast to all address.

Parameters **udp_address** (str) – the new UDP address

Return type str

Returns the command string & UDP address set

simulator_thread_running()
Return the state of the PhaseMicroStepper simulator thread.

Return type bool

Returns True if the simulator thread is running

start_communicating()
Establish communication with the phasemicrostepper.

Return type None

stop_communicating()
Stop communicating with the PhaseMicroStepper.

Return type None

sync_pps()
Shift the rising edge of 1PPS OUT signals in order to align it with 1PPS IN.

The value in the second string is 0 if the operation is OK, 1 in case of error. In most of the cases, the error is due to the absence of a valid signal on the 1PPS IN connector.

Returns the command string and second string as "syncPPS done, Error = 0"

class PhaseMicroStepperHealthModel(*health_changed_callback*)
A health model for a phase microstepper.

At present this uses the base health model; this is a placeholder for a future, better implementation.

class SatPhaseMicroStepper(**args*, ***kwargs*)
An implementation of a PhaseMicroStepper Tango device for SatLmc.

Advance10M(*argin*)
Advance the output phase of 10MHz OUT signals, in units of 10ns.

Acceptable range is from 0 to +9.

Parameters **argin** (int) – the phase advance in units of 10ns

Return type typing.Tuple[typing.List[ska_tango_base.commands.ResultCode], typing.List[typing.Optional[str]]]

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

class Advance10MCommand(**args*, ***kwargs*)
Class for handling the Advance10M command.

do(*argin*)
Implement Advance10M command functionality.
Parameters **argin** (int) – the advance in units of 10ns
Return type typing.Tuple[ska_tango_base.commands.ResultCode, str]
Returns A tuple containing a return code and a string message indicating status. The message is for information purpose only.

AdvanceAllPhase(*argin*)
Advance the output phase of all OUT signals, in units of 1e-15s.

The resolution is about 1e-13. Acceptable range is from -5000000 to +5000000.

Parameters **argin** (int) – the advance in units of 1e-15s

Return type typing.Tuple[typing.List[ska_tango_base.commands.ResultCode], typing.List[typing.Optional[str]]]

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

class AdvanceAllPhaseCommand(**args*, ***kwargs*)
Class for handling the AdvanceAllPhase command.

do(*argin*)
Implement AdvanceAllPhase command functionality.
Parameters **argin** (int) – the advance in units of 1e-15s
Return type typing.Tuple[ska_tango_base.commands.ResultCode, str]
Returns A tuple containing a return code and a string message indicating status. The message is for information purpose only.

AdvancePPS(*argin*)
Advance the phase of PPS OUT signals in units of 10ns.

The acceptable range is limited from -99000000 to 30000000.AdvancePPS.

Parameters **argin** (int) – the advance in units of 10ns

Return type typing.Tuple[typing.List[ska_tango_base.commands.ResultCode], typing.List[typing.Optional[str]]]

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

class AdvancePPSCommand(**args*, ***kwargs*)
Class for handling the AdvancePPS command.

do(*argin*)
Implement AdvancePPS command functionality.
Parameters **argin** (int) – the advance in units of 10ns
Return type typing.Tuple[ska_tango_base.commands.ResultCode, str]
Returns A tuple containing a return code and a string message indicating status. The message is for information purpose only.

GetOffsetFrequency()
Get the offset frequency, in units of 1e-20.

Return type str

Returns the offset frequency send in the last setOffsetFrequencu command.

class GetOffsetFrequencyCommand(**args*, ***kwargs*)
Class for handling the GetOffsetFrequency() command.

do()
Implement GetOffsetFrequency() command functionality.
Return type str
Returns the offset frequency

class InitCommand(**args*, ***kwargs*)
Implement the device initialisation for the PhaseMicroStepper device.

do()
Initialise the attributes and properties.
Returns A tuple containing a return code and a string message indicating status. The message is for information purpose only.

class SetIPAddressCommand(**args*, ***kwargs*)
Class for handling the SetIPAddress command.

do(*argin*)
Implement SetIPAddress command functionality.
Parameters **argin** (str) – the new IP address
Return type typing.Tuple[ska_tango_base.commands.ResultCode, str]
Returns A tuple containing a return code and a string message indicating status. The message is for information purpose only.

SetIPaddress(*argin*)
Set the IP address of the EOG and closes the TCP connection.

Parameters **argin** (str) – the new IP address

Return type typing.Tuple[typing.List[ska_tango_base.commands.ResultCode], typing.List[typing.Optional[str]]]

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

SetOffsetFrequency(*offset*)
Set the offset frequency.

Set the offset frequency of OUT signals respect to the frequency of 100MHz IN signal, in units of 1e-20. 7 Resolution is better than 3e-20. The acceptable range is from -1000000000000 to 1000000000000

Parameters **offset** (int) – the offset frequency

Return type typing.Tuple[typing.List[ska_tango_base.commands.ResultCode], typing.List[typing.Optional[str]]]

Returns the command + the converted offset frequency

class SetOffsetFrequencyCommand(**args*, ***kwargs*)
Class for handling the SetOffsetFrequency(argin) command.

do(*argin*)
Implement SetOffsetFrequency command functionality.
Parameters **argin** (int) – the offset frequency
Return type typing.Tuple[ska_tango_base.commands.ResultCode, str]
Returns A tuple containing a return code and a string message indicating status. The message is for information purpose only.

class SetUDPAddressCommand(**args*, ***kwargs*)
Class for handling the SetUDPAddress command.

do(*argin*)
Implement SetUDPAddress command functionality.

Parameters **argin** (str) – the new UDP address
Return type typing.Tuple[ska_tango_base.commands.ResultCode, str]
Returns A tuple containing a return code and a string message indicating status. The message is for information purpose only.

SetUDPaddress(*argin*)
Set the UDP address destination.

Parameters **argin** (str) – the str

Return type typing.Tuple[typing.List[ska_tango_base.commands.ResultCode], typing.List[typing.Optional[str]]]

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

SyncPPS()
External sync of the 1PPS outputs.

Return type typing.Tuple[typing.List[ska_tango_base.commands.ResultCode], typing.List[typing.Optional[str]]]

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

class SyncPPSCommand(**args*, ***kwargs*)
Class for handling the SyncPPS(argin) command.

do()
Implement SyncPPS command functionality.
Return type typing.Tuple[ska_tango_base.commands.ResultCode, str]
Returns A tuple containing a return code and a string message indicating status. The message is for information purpose only.

create_component_manager()
Create and return a component manager for this device.

Return type *ska_sat_lmc.phasemicrostepper.phasemicrostepper_driver.PhaseMicroStepperDriver*

Returns a component manager for this device.

first_internal_oscillator()
Report the status of the first internal oscillator.

Return type float

Returns oscillator 1 value (fixed point 6 decimal digits)

health_changed(*health*)
Handle change in this device's health state.

This is a callback hook, called whenever the HealthModel's evaluated health state changes. It is responsible for updating the tango side of things i.e. making sure the attribute is up to date, and events are pushed.

Parameters **health** (ska_tango_base.control_model.HealthState) – the new health value

Return type None

init_command_objects()
Initialise the command handlers for commands supported by this device.

Return type None

init_device()
Initialise the device.

This is overridden here to change the Tango serialisation model.

Return type None

internal_temperature()
Report the internal temperature of the devicec.

Return type float

Returns temperature in celcius (fixed point 2 decimal digits)

is_Advance10M_allowed()
Check if command Advance10M is allowed in the current device state.

Return type bool

Returns True if the command is allowed

is_AdvanceAllPhase_allowed()
Check if command AdvanceAllPhase is allowed in the current device state.

Return type bool

Returns True if the command is allowed

is_AdvancePPS_allowed()
Check if command *AdvancePPS* is allowed in the current device state.

Return type bool

Returns True if the command is allowed

is_SetIPaddress_allowed()
Check if command *SetIPaddress* is allowed in the current device state.

Return type bool

Returns True if the command is allowed

is_SetUDPaddress_allowed()
Check if command *SetUDPaddress* is allowed in the current device state.

Return type bool

Returns True if the command is allowed

is_SyncPPS_allowed()
Check if command *SyncPPS* is allowed in the current device state.

Return type bool

Returns True if the command is allowed

is_attribute_allowed(*attr_req_type*)
Protect attribute access before being updated otherwise it reports alarm.

Parameters **attr_req_type** (tango.AttReqType) – tango attribute type READ/WRITE

Return type bool

Returns True if the attribute can be read else False

second_internal_oscillator()
Report the status of the second internal oscillator.

Return type float

Returns oscillator 2 value (fixed point 6 decimal digits)

serial_number()
Report the serial number of the device.

Return type int

Returns the serial number

simulationMode(*value*)
Set the simulation mode.

Parameters **value** (ska_tango_base.control_model.SimulationMode) – The simulation mode, as a SimulationMode value

Return type None

testMode(*value*)
Set the test mode.

Parameters **value** (ska_tango_base.control_model.TestMode) – The test mode, as a TestMode value

Return type None

1.3.1 Phase Micro Stepper Component Manager

This module implements phasemicrostepper component management.

class PhaseMicroStepperComponentManager(*initial_simulation_mode*, *initial_test_mode*, *logger*, *phase_url*, *simulator_url*, *communication_status_changed_callback*, *component_fault_callback*)
A component manager that handles a PhaseMicroStepper simulator and driver.

__init__(*initial_simulation_mode*, *initial_test_mode*, *logger*, *phase_url*, *simulator_url*, *communication_status_changed_callback*, *component_fault_callback*)
Initialise a new instance.

Parameters

- **initial_simulation_mode** (ska_tango_base.control_model.SimulationMode) – the simulation mode that the component should start in
- **initial_test_mode** (ska_tango_base.control_model.TestMode) – the simulation mode that the component should start in
- **logger** (logging.Logger) – a logger for this object to use
- **phase_url** (str) – the URL of the PhaseMicroStepper
- **simulator_url** (str) – the URL of the simulated PhaseMicroStepper
- **communication_status_changed_callback** (typing.Callable[[*ska_sat_lmc.component.component_manager.CommunicationStatus*], None]) – callback to be called when the status of the communications channel between the component manager and its component changes
- **component_fault_callback** (typing.Callable[[bool], None]) – callback to be called when the component faults (or stops faulting)

property simulation_mode: ska_tango_base.control_model.SimulationMode
Return the simulation mode.

Return type ska_tango_base.control_model.SimulationMode

Returns the simulation mode

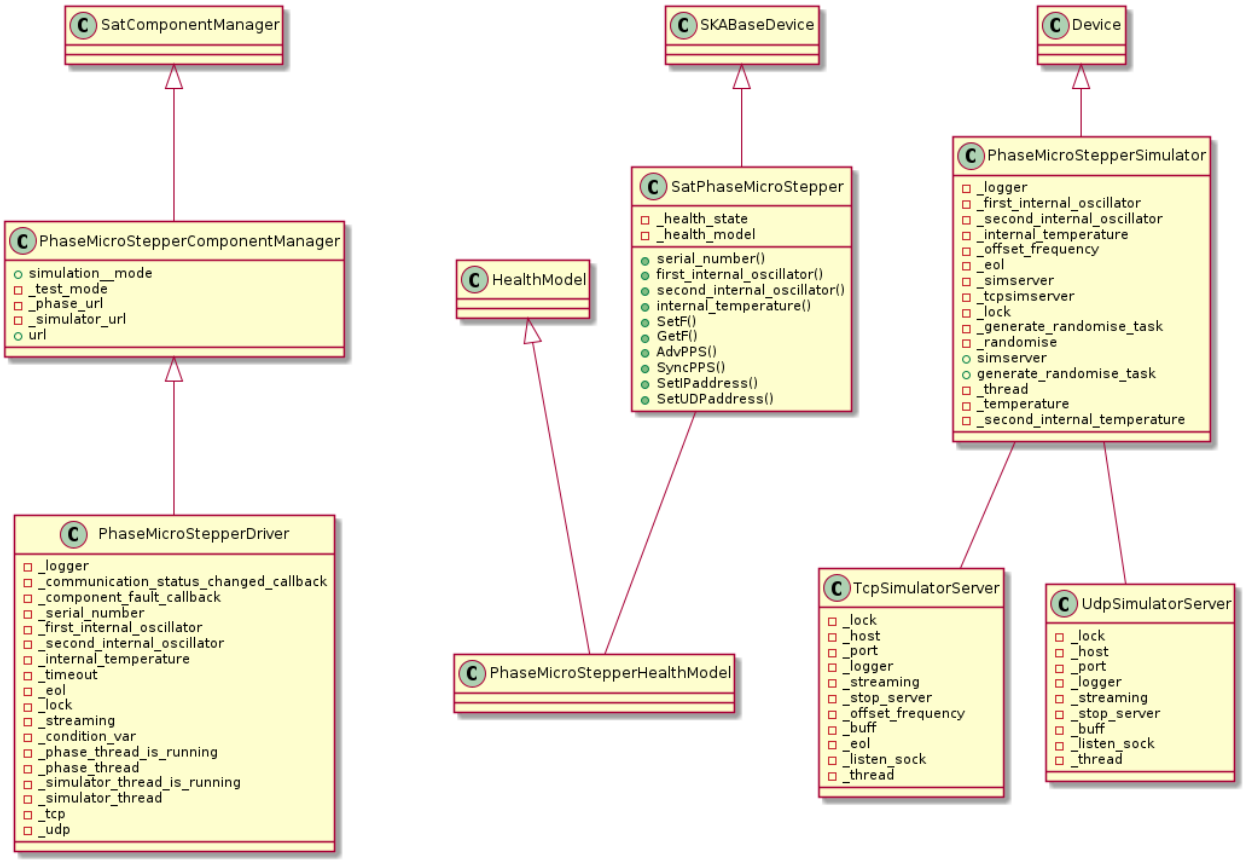
property test_mode: ska_tango_base.control_model.TestMode

Return the test mode.

Return type ska_tango_base.control_model.TestMode

Returns the test mode

1.3.2 Phase Micro Stepper Device



This module implements the SATPhaseMicroStepper device.

class SatPhaseMicroStepper(**args*, ***kwargs*)

An implementation of a PhaseMicroStepper Tango device for SatLmc.

Advance10M(*argin*)

Advance the output phase of 10MHz OUT signals, in units of 10ns.

Acceptable range is from 0 to +9.

Parameters **argin** (int) – the phase advance in units of 10ns

Return type typing.Tuple[typing.List[ska_tango_base.commands.ResultCode], typing.List[typing.Optional[str]]]

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

class Advance10MCommand(**args*, ***kwargs*)
Class for handling the Advance10M command.

do(*argin*)
Implement Advance10M command functionality.
Parameters **argin** (int) – the advance in units of 10ns
Return type typing.Tuple[ska_tango_base.commands.ResultCode, str]
Returns A tuple containing a return code and a string message indicating status. The message is for information purpose only.

AdvanceAllPhase(*argin*)
Advance the output phase of all OUT signals, in units of 1e-15s.

The resolution is about 1e-13. Acceptable range is from -5000000 to +5000000.

Parameters **argin** (int) – the advance in units of 1e-15s

Return type typing.Tuple[typing.List[ska_tango_base.commands.ResultCode], typing.List[typing.Optional[str]]]

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

class AdvanceAllPhaseCommand(**args*, ***kwargs*)
Class for handling the AdvanceAllPhase command.

do(*argin*)
Implement AdvanceAllPhase command functionality.
Parameters **argin** (int) – the advance in units of 1e-15s
Return type typing.Tuple[ska_tango_base.commands.ResultCode, str]
Returns A tuple containing a return code and a string message indicating status. The message is for information purpose only.

AdvancePPS(*argin*)
Advance the phase of PPS OUT signals in units of 10ns.

The acceptable range is limited from -99000000 to 30000000.AdvancePPS.

Parameters **argin** (int) – the advance in units of 10ns

Return type typing.Tuple[typing.List[ska_tango_base.commands.ResultCode], typing.List[typing.Optional[str]]]

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

class AdvancePPSCommand(**args*, ***kwargs*)
Class for handling the AdvancePPS command.

do(*argin*)
Implement AdvancePPS command functionality.
Parameters **argin** (int) – the advance in units of 10ns
Return type typing.Tuple[ska_tango_base.commands.ResultCode, str]
Returns A tuple containing a return code and a string message indicating status. The message is for information purpose only.

GetOffsetFrequency()
Get the offset frequency, in units of 1e-20.

Return type str

Returns the offset frequency send in the last setOffsetFrequencu command.

class GetOffsetFrequencyCommand(*args, **kwargs)
Class for handling the GetOffsetFrequency() command.

do()
Implement GetOffsetFrequency() command functionality.
Return type str
Returns the offset frequency

class InitCommand(*args, **kwargs)
Implement the device initialisation for the PhaseMicroStepper device.

do()
Initialise the attributes and properties.
Returns A tuple containing a return code and a string message indicating status. The message is for information purpose only.

class SetIPAddressCommand(*args, **kwargs)
Class for handling the SetIPAddress command.

do(*argin*)
Implement SetIPAddress command functionality.
Parameters **argin** (str) – the new IP address
Return type typing.Tuple[ska_tango_base.commands.ResultCode, str]
Returns A tuple containing a return code and a string message indicating status. The message is for information purpose only.

SetIPaddress(*argin*)
Set the IP address of the EOG and closes the TCP connection.

Parameters **argin** (str) – the new IP address

Return type typing.Tuple[typing.List[ska_tango_base.commands.ResultCode], typing.List[typing.Optional[str]]]

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

SetOffsetFrequency(*offset*)
Set the offset frequency.

Set the offset frequency of OUT signals respect to the frequency of 100MHz IN signal, in units of 1e-20. 7 Resolution is better than 3e-20. The acceptable range is from -1000000000000 to 1000000000000

Parameters **offset** (int) – the offset frequency

Return type typing.Tuple[typing.List[ska_tango_base.commands.ResultCode], typing.List[typing.Optional[str]]]

Returns the command + the converted offset frequency

class SetOffsetFrequencyCommand(*args, **kwargs)
Class for handling the SetOffsetFrequency(argin) command.

do(*argin*)
Implement SetOffsetFrequency command functionality.
Parameters **argin** (int) – the offset frequency
Return type typing.Tuple[ska_tango_base.commands.ResultCode, str]
Returns A tuple containing a return code and a string message indicating status. The message is for information purpose only.

class SetUDPAddressCommand(*args, **kwargs)
Class for handling the SetUDPAddress command.

do(*argin*)
Implement SetUDPAddress command functionality.
Parameters **argin** (str) – the new UDP address
Return type typing.Tuple[ska_tango_base.commands.ResultCode, str]
Returns A tuple containing a return code and a string message indicating status. The message is for information purpose only.

SetUDPaddress(*argin*)
Set the UDP address destination.

Parameters **argin** (str) – the str

Return type typing.Tuple[typing.List[ska_tango_base.commands.ResultCode], typing.List[typing.Optional[str]]]

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

SyncPPS()
External sync of the 1PPS outputs.

Return type typing.Tuple[typing.List[ska_tango_base.commands.ResultCode], typing.List[typing.Optional[str]]]

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

class SyncPPSCommand(**args*, ***kwargs*)
Class for handling the SyncPPS(argin) command.

do()
Implement SyncPPS command functionality.
Return type typing.Tuple[ska_tango_base.commands.ResultCode, str]
Returns A tuple containing a return code and a string message indicating status. The message is for information purpose only.

create_component_manager()
Create and return a component manager for this device.

Return type *ska_sat_lmc.phasemicrostepper.phasemicrostepper_driver.PhaseMicroStepperDriver*

Returns a component manager for this device.

first_internal_oscillator()
Report the status of the first internal oscillator.

Return type float

Returns oscillator 1 value (fixed point 6 decimal digits)

health_changed(*health*)
Handle change in this device's health state.

This is a callback hook, called whenever the HealthModel's evaluated health state changes. It is responsible for updating the tango side of things i.e. making sure the attribute is up to date, and events are pushed.

Parameters **health** (ska_tango_base.control_model.HealthState) – the new health value

Return type None

init_command_objects()
Initialise the command handlers for commands supported by this device.

Return type None

init_device()
Initialise the device.

This is overridden here to change the Tango serialisation model.

Return type None

internal_temperature()
Report the internal temperature of the devicec.

Return type float

Returns temperature in celcius (fixed point 2 decimal digits)

is_Advance10M_allowed()
Check if command Advance10M is allowed in the current device state.

Return type bool

Returns True if the command is allowed

is_AdvanceAllPhase_allowed()
Check if command AdvanceAllPhase is allowed in the current device state.

Return type bool

Returns True if the command is allowed

is_AdvancePPS_allowed()
Check if command *AdvancePPS* is allowed in the current device state.

Return type bool

Returns True if the command is allowed

is_SetIPaddress_allowed()
Check if command *SetIPaddress* is allowed in the current device state.

Return type bool

Returns True if the command is allowed

is_SetUDPaddress_allowed()
Check if command *SetUDPaddress* is allowed in the current device state.

Return type bool

Returns True if the command is allowed

is_SyncPPS_allowed()
Check if command *SyncPPS* is allowed in the current device state.

Return type bool

Returns True if the command is allowed

is_attribute_allowed(*attr_req_type*)
Protect attribute access before being updated otherwise it reports alarm.

Parameters **attr_req_type** (tango.AttReqType) – tango attribute type READ/WRITE

Return type bool

Returns True if the attribute can be read else False

second_internal_oscillator()
Report the status of the second internal oscillator.

Return type float

Returns oscillator 2 value (fixed point 6 decimal digits)

serial_number()
Report the serial number of the device.

Return type int

Returns the serial number

simulationMode(*value*)
Set the simulation mode.

Parameters **value** (ska_tango_base.control_model.SimulationMode) – The simulation mode, as a SimulationMode value

Return type None

testMode(*value*)
Set the test mode.

Parameters **value** (ska_tango_base.control_model.TestMode) – The test mode, as a TestMode value

Return type None

main(**args*, ***kwargs*)
Entry point for module.

Parameters

- **args** (str) – positional arguments
- **kwargs** (str) – named arguments

Return type int

Returns exit code

1.3.3 Phase Micro Stepper Driver

This module implements the PhaseMicrostepper driver.

class PhaseMicroStepperDriver(*initial_simulation__mode*, *initial_test_mode*, *logger*, *phase_url*, *simulator_url*, *communication_status_changed_callback*, *component_fault_callback*)
Device Server for the SAT.LMC PhaseMicrostepper (100MHz).

__init__(*initial_simulation__mode*, *initial_test_mode*, *logger*, *phase_url*, *simulator_url*, *communication_status_changed_callback*, *component_fault_callback*)
Initialise the attributes and properties of the PhaseMicroStepper.

Parameters

- **initial_simulation__mode** (ska_tango_base.control_model.SimulationMode) – the simulation mode that the component should start in
- **initial_test_mode** (ska_tango_base.control_model.TestMode) – the simulation mode that the component should start in
- **logger** (logging.Logger) – a logger for this object to use

- **phase_url** (str) – the URL of the PhaseMicroStepper
- **simulator_url** (str) – the URL of the simulated PhaseMicroStepper
- **communication_status_changed_callback** (typing.Callable[[*ska_sat_lmc.component.component_manager.CommunicationStatus*], None]) – callback to be called when the status of the communications channel between the component manager and its component changes
- **component_fault_callback** (typing.Callable[[bool], None]) – callback to be called when the component faults (or stops faulting)

advance_10megahertz(*advance*)

Advance the output phase of 10MHz OUT signals.

Value in units of 10ns. Acceptable range is from 0 to +9.

Parameters **advance** (int) – advance by this value

Return type str

Returns the command string & value

advance_phase(*advance*)

Advance the output phase of all OUT signals.in units of 1e-15s.

The resolution is about 1e-13. Acceptable range is from -5000000 to +5000000.

Parameters **advance** (int) – advance by this value

Return type str

Returns the command string & value

advance_pps(*advance*)

Advance the phase of PPS OUT signals in units of 10ns.

The acceptable range is limited from -99000000 to 30000000.

Parameters **advance** (int) – advance by this value

Return type str

Returns the command string & value

property first_internal_oscillator: float

Return the status of internal oscillator 1.

value 1 is fixed point 6 decimal digits

Return type float

Returns the status of internal oscillator 1

get_offset_frequency()

Return the offset frequency set in the last setF command.

Return type str

Returns the command and offset frequency in units of 1e-20

property internal_temperature: float

Return the status of internal Celsius temperature.

value 3 is fixed point 2 decimal digits.

Return type float

Returns the internal temperature

phase_thread_running()
Return the state of the PhaseMicroStepper hardware thread.

Return type bool

Returns True if the PhaseMicroStepper thread is running

property second_internal_oscillator: float
Return the status of internal oscillator 2.

value 2 is fixed point 6 decimal digits.

Return type float

Returns the status of internal oscillator 2

property serial_number: int
Return the serial number of the EOG.

Return type int

Returns the serial number

set_ip_address(*ip_address*)
Set the IP address of the EOG and closes the TCP connection.

Parameters **ip_address** (str) – the new IP address

Return type str

Returns the command string & IP address set

set_offset_frequency(*offset*)
Set the offset frequency.

Sets the OUT signals respect to the frequency of 100MHz IN signal, in units of 1e-20. Resolution is better than 3e-20. The acceptable range is from -1000000000000 to 1000000000000.

Parameters **offset** (int) – the offset frequency

Return type str

Returns the command and converted frquency in units of 1e-20

set_udp_address(*udp_address*)
Set the UDP destination address.

If network address is 255.255.255.255, the UDP message is broadcast to all address.

Parameters **udp_address** (str) – the new UDP address

Return type str

Returns the command string & UDP address set

simulator_thread_running()
Return the state of the PhaseMicroStepper simulator thread.

Return type bool

Returns True if the simulator thread is running

start_communicating()
Establish communication with the phasemicrostepper.

Return type None

stop_communicating()
Stop communicating with the PhaseMicroStepper.

Return type None

sync_pps()
Shift the rising edge of 1PPS OUT signals in order to align it with 1PPS IN.

The value in the second string is 0 if the operation is OK, 1 in case of error. In most of the cases, the error is due to the absence of a valid signal on the 1PPS IN connector.

Returns the command string and second string as "syncPPS done, Error = 0"

1.3.4 Phase Micro Stepper Health Model

An implementation of a health model for a phase microstepper.

class PhaseMicroStepperHealthModel(*health_changed_callback*)
A health model for a phase microstepper.

At present this uses the base health model; this is a placeholder for a future, better implementation.

1.4 Component subpackage

This module implements infrastructure for component management in SAT.LMC.

class CommunicationStatus(*value*)
The status of a component manager's communication with its component.

DISABLED = 1
The component manager is not trying to establish/maintain a channel of communication with its component. For example:

- if communication with the component is connection-oriented, then there is no connection, and the component manager is not trying to establish a connection.
- if communication with the component is by event subscription, then the component manager is unsubscribed from events.
- if communication with the component is by periodic connectionless polling, then the component manager is not performing that polling.

ESTABLISHED = 3
The component manager has established a channel of communication with its component. For example:

- if communication with the component is connection-oriented, then the component manager has connected to its component.

NOT_ESTABLISHED = 2
The component manager is trying to establish/maintain a channel of communication with its component, but that channel is not currently established. For example:

- if communication with the component is connection-oriented, then the component manager has failed to establish/maintain the connection.

class DeviceComponentManager(*fqdn*, *logger*, *communication_status_changed_callback*, *component_fault_callback*, *health_changed_callback=None*)
An abstract component manager for a Tango device component.

__init__(*fqdn*, *logger*, *communication_status_changed_callback*, *component_fault_callback*, *health_changed_callback=None*)

Initialise a new instance.

Parameters

- **fqdn** (str) – the FQDN of the device
- **logger** (logging.Logger) – the logger to be used by this object.
- **communication_status_changed_callback** (typing.Callable[[*ska_sat_lmc.component.component_manager.CommunicationStatus*], None]) – callback to be called when the status of the communications channel between the component manager and its component changes
- **component_fault_callback** (typing.Optional[typing.Callable[[bool], None]]) – callback to be called when the component faults (or stops faulting)
- **health_changed_callback** (typing.Optional[typing.Callable[[typing.Optional[ska_tango_base.control_model.HealthState]], None]]) – callback to be called when the health state of the device changes. The value it is called with will normally be a HealthState, but may be None if the admin mode of the device indicates that the device's health should not be included in upstream health rollup.

property health: Optional[ska_tango_base.control_model.HealthState]

Return the evaluated health state of the device.

This will be either the health state that the device reports, or None if the device is in an admin mode that indicates that its health should not be rolled up.

Return type typing.Optional[ska_tango_base.control_model.HealthState]

Returns the evaluated health state of the device.

start_communicating()

Establish communication with the component, then start monitoring.

This is a public method that enqueues the work to be done.

Return type None

stop_communicating()

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

Return type None

class SatComponentManager(*logger*, *communication_status_changed_callback*, *component_fault_callback*, **args*, ***kwargs*)

A base component manager for Sat LMC.

This class exists to modify the interface of the ska_tango_base.base.component_manager.BaseComponentManager. The `BaseComponentManager` accepts an `op_state_model` argument, and is expected to interact directly with it. This is not a very good design decision. It is better to leave the ``op_state_model` behind in the device, and drive it indirectly through callbacks.

Therefore this class accepts two callback arguments: one for when communication with the component changes and one for when the component fault status changes. In the last case, callback hooks are provided so that the component can indicate the change to this component manager.

__init__(*logger*, *communication_status_changed_callback*, *component_fault_callback*, **args*, ***kwargs*)

Initialise a new instance.

Parameters

- **logger** (logging.Logger) – a logger for this instance to use
- **communication_status_changed_callback** (typing.Optional[typing.Callable[[ska_sat_lmc.component.component_manager.CommunicationStatus], None]]) – callback to be called when the status of communications between the component manager and its component changes.
- **component_fault_callback** (typing.Optional[typing.Callable[[bool], None]]) – callback to be called when the fault status of the component changes.
- **args** (typing.Any) – other positional args
- **kwargs** (typing.Any) – other keyword args

property communication_status: ska_sat_lmc.component.component_manager.CommunicationStatus

Return the communication status of this component manager.

This is implemented as a replacement for the `is_communicating` property, which should be deprecated.

Return type *ska_sat_lmc.component.component_manager.CommunicationStatus*

Returns status of the communication channel with the component.

component_fault_changed(*faulty*)

Handle notification that the component's fault status has changed.

This is a callback hook, to be passed to the managed component.

Parameters **faulty** (bool) – whether the component has faulted. If `False`, then this is a notification that the component has *recovered* from a fault.

Return type None

property faulty: Optional[bool]

Return whether this component manager is currently experiencing a fault.

Return type typing.Optional[bool]

Returns whether this component manager is currently experiencing a fault.

property is_communicating: bool

Return communication with the component is established.

SatLmc uses the more expressive *communication_status* for this, but this is still needed as a base classes hook.

Return type bool

Returns whether communication with the component is established.

start_communicating()

Start communicating with the component.

Return type None

stop_communicating()

Break off communicating with the component.

Return type None

update_communication_status(*communication_status*)

Handle a change in communication status.

This is a helper method for use by subclasses.

Parameters **communication_status** (*ska_sat_lmc.component.component_manager.CommunicationStatus*) – the new communication status of the component manager.

Return type None

update_component_fault(*faulty*)
Update the component fault status, calling callbacks as required.

This is a helper method for use by subclasses.

Parameters **faulty** (typing.Optional[bool]) – whether the component has faulted. If False, then this is a notification that the component has *recovered* from a fault.

Return type None

check_communicating(*func*)
Return a function that checks component communication before calling a function.

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

This function is intended to be used as a decorator:

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

Parameters **func** (typing.TypeVar(Wrapped, bound= typing.Callable[..., typing.Any])) – the wrapped function

Return type typing.TypeVar(Wrapped, bound= typing.Callable[..., typing.Any])

Returns the wrapped function

1.4.1 Component Manager

This module implements a functionality for component managers in SAT.LMC.

class CommunicationStatus(*value*)
The status of a component manager's communication with its component.

DISABLED = 1
The component manager is not trying to establish/maintain a channel of communication with its component. For example:

- if communication with the component is connection-oriented, then there is no connection, and the component manager is not trying to establish a connection.
- if communication with the component is by event subscription, then the component manager is unsubscribed from events.
- if communication with the component is by periodic connectionless polling, then the component manager is not performing that polling.

ESTABLISHED = 3
The component manager has established a channel of communication with its component. For example:

- if communication with the component is connection-oriented, then the component manager has connected to its component.

NOT_ESTABLISHED = 2
The component manager is trying to establish/maintain a channel of communication with its component, but that channel is not currently established. For example:

- if communication with the component is connection-oriented, then the component manager has failed to establish/maintain the connection.

class SatComponentManager(*logger*, *communication_status_changed_callback*, *component_fault_callback*, **args*, ***kwargs*)
A base component manager for Sat LMC.

This class exists to modify the interface of the ska_tango_base.base.component_manager.BaseComponentManager. The BaseComponentManager accepts an op_state_model` argument, and is expected to interact directly with it. This is not a very good design decision. It is better to leave the ``op_state_model behind in the device, and drive it indirectly through callbacks.

Therefore this class accepts two callback arguments: one for when communication with the component changes and one for when the component fault status changes. In the last case, callback hooks are provided so that the component can indicate the change to this component manager.

__init__(*logger*, *communication_status_changed_callback*, *component_fault_callback*, **args*, ***kwargs*)
Initialise a new instance.

Parameters

- **logger** (logging.Logger) – a logger for this instance to use
- **communication_status_changed_callback** (typing.Optional[typing.Callable[[*ska_sat_lmc.component.component_manager.CommunicationStatus*], None]]) – callback to be called when the status of communications between the component manager and its component changes.
- **component_fault_callback** (typing.Optional[typing.Callable[[bool], None]]) – callback to be called when the fault status of the component changes.
- **args** (typing.Any) – other positional args
- **kwargs** (typing.Any) – other keyword args

property communication_status: ska_sat_lmc.component.component_manager.CommunicationStatus
Return the communication status of this component manager.

This is implemented as a replacement for the is_communicating property, which should be deprecated.

Return type *ska_sat_lmc.component.component_manager.CommunicationStatus*

Returns status of the communication channel with the component.

component_fault_changed(*faulty*)
Handle notification that the component's fault status has changed.

This is a callback hook, to be passed to the managed component.

Parameters **faulty** (bool) – whether the component has faulted. If False, then this is a notification that the component has *recovered* from a fault.

Return type None

property faulty: Optional[bool]
Return whether this component manager is currently experiencing a fault.

Return type typing.Optional[bool]

Returns whether this component manager is currently experiencing a fault.

property is_communicating: bool
Return communication with the component is established.

SatLmc uses the more expressive *communication_status* for this, but this is still needed as a base classes hook.

Return type bool

Returns whether communication with the component is established.

start_communicating()
Start communicating with the component.

Return type None

stop_communicating()
Break off communicating with the component.

Return type None

update_communication_status(*communication_status*)
Handle a change in communication status.

This is a helper method for use by subclasses.

Parameters **communication_status** (*ska_sat_lmc.component.component_manager.CommunicationStatus*) – the new communication status of the component manager.

Return type None

update_component_fault(*faulty*)
Update the component fault status, calling callbacks as required.

This is a helper method for use by subclasses.

Parameters **faulty** (typing.Optional[bool]) – whether the component has faulted. If False, then this is a notification that the component has *recovered* from a fault.

Return type None

1.4.2 Device Component Manager

This module implements an abstract component manager for simple object components.

class DeviceComponentManager(*fqdn*, *logger*, *communication_status_changed_callback*, *component_fault_callback*, *health_changed_callback=None*)
An abstract component manager for a Tango device component.

__init__(*fqdn*, *logger*, *communication_status_changed_callback*, *component_fault_callback*, *health_changed_callback=None*)
Initialise a new instance.

Parameters

- **fqdn** (str) – the FQDN of the device
- **logger** (logging.Logger) – the logger to be used by this object.
- **communication_status_changed_callback** (typing.Callable[[*ska_sat_lmc.component.component_manager.CommunicationStatus*], None]) – callback to be called when the status of the communications channel between the component manager and its component changes

- **component_fault_callback** (typing.Optional[typing.Callable[[bool], None]]) – callback to be called when the component faults (or stops faulting)
- **health_changed_callback** (typing.Optional[typing.Callable[[typing.Optional[ska_tango_base.control_model.HealthState]], None]]) – callback to be called when the health state of the device changes. The value it is called with will normally be a HealthState, but may be None if the admin mode of the device indicates that the device's health should not be included in upstream health rollup.

property health: Optional[ska_tango_base.control_model.HealthState]
Return the evaluated health state of the device.

This will be either the health state that the device reports, or None if the device is in an admin mode that indicates that its health should not be rolled up.

Return type typing.Optional[ska_tango_base.control_model.HealthState]

Returns the evaluated health state of the device.

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

This is a public method that enqueues the work to be done.

Return type None

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

Return type None

1.4.3 Util

This module implements utils for component managers in SAT.LMC.

check_communicating(*func*)
Return a function that checks component communication before calling a function.

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

This function is intended to be used as a decorator:

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

Parameters **func** (typing.TypeVar(Wrapped, bound= typing.Callable[..., typing.Any])) – the wrapped function

Return type typing.TypeVar(Wrapped, bound= typing.Callable[..., typing.Any])

Returns the wrapped function

1.5 Testing subpackage

This subpackage contains modules for helper classes in the SKA SAT LMC tests.

class TangoHarness(**args*, ***kwargs*)
Abstract base class for Tango test harnesses.

This does very little, because it needs to support both harnesses that directly interact with Tango, and wrapper harnesses that add functionality to another harness.

The one really important thing it does do, is ensure that *ska_sat_lmc.device_proxy.SatDeviceProxy* uses this harness's `connection factory` to make connections.

__init__(**args*, ***kwargs*)
Initialise a new instance.

Parameters

- **args** (typing.Any) – additional positional arguments
- **kwargs** (typing.Any) – additional keyword arguments

property connection_factory: Callable[[str], tango.DeviceProxy]
Establish connections to devices with this factory.

Raises NotImplementedError – because this method is abstract

Return type typing.Callable[[str], tango.DeviceProxy]

property fqdns: list[str]
Return FQDNs of devices in this harness.

Raises NotImplementedError – because this method is abstract

get_device(*fqdn*)
Create and return a proxy to the device at the given FQDN.

Parameters **fqdn** (str) – FQDN of the device for which a proxy is required

Raises NotImplementedError – because this method is abstract

Return type *ska_sat_lmc.device_proxy.SatDeviceProxy*

1.5.1 Tango harness

This module implements a SatLMC test harness for Tango devices.

class BaseTangoHarness(*device_info*, *logger*, **args*, ***kwargs*)
A basic test harness for Tango devices.

This harness doesn't stand up any device; it assumes that devices are already running. It is thus useful for testing against deployed devices.

__init__(*device_info*, *logger*, **args*, ***kwargs*)
Initialise a new instance.

Parameters

- **device_info** (typing.Optional[*ska_sat_lmc.testing.tango_harness.SatDeviceInfo*]) – object that makes device info available
- **logger** (logging.Logger) – a logger for the harness
- **args** (typing.Any) – additional positional arguments

- **kwargs** (typing.Any) – additional keyword arguments

property connection_factory: Callable[[str], tango.DeviceProxy]
Establish connections to devices with this factory.

This class uses tango.DeviceProxy as its connection factory.

Return type typing.Callable[[str], tango.DeviceProxy]

Returns a DeviceProxy for use in establishing connections.

property fqdns: list[str]
Return the FQDNs of devices in this harness.

Returns a list of FQDNs of devices in this harness.

get_device(*fqdn*)
Create and return a proxy to the device at the given FQDN.

Parameters **fqdn** (str) – FQDN of the device for which a proxy is required

Return type *ska_sat_lmc.device_proxy.SatDeviceProxy*

Returns A proxy of the type specified by the proxy map.

class ClientProxyTangoHarness(*device_info*, *logger*, **args*, ***kwargs*)
A test harness for Tango devices that can return tailored client proxies.

__init__(*device_info*, *logger*, **args*, ***kwargs*)
Initialise a new instance.

Parameters

- **device_info** (typing.Optional[*ska_sat_lmc.testing.tango_harness.SatDeviceInfo*]) – object that makes device info available
- **logger** (logging.Logger) – a logger for the harness
- **args** (typing.Any) – additional positional arguments
- **kwargs** (typing.Any) – additional keyword arguments

get_device(*fqdn*)
Create and return a proxy to the device at the given FQDN.

Parameters **fqdn** (str) – FQDN of the device for which a proxy is required

Return type *ska_sat_lmc.device_proxy.SatDeviceProxy*

Returns A proxy of the type specified by the proxy map.

class MockingTangoHarness(*harness*, *mock_factory*, *initial_mocks*, **args*, ***kwargs*)
A Tango test harness that mocks out devices not under test.

This harness wraps another harness, but only uses that harness for a specified set of devices under test, and mocks out all others.

__init__(*harness*, *mock_factory*, *initial_mocks*, **args*, ***kwargs*)
Initialise a new instance.

Parameters

- **harness** – the wrapped harness
- **mock_factory** – the factory to be used to build mocks
- **initial_mocks** – a pre-build dictionary of mocks to be used for particular

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

property connection_factory: Callable[[str], tango.DeviceProxy]
Establish connections to devices with this factory.

This is where we check whether the requested device is on our list. Devices on the list are passed to the connection factory of the wrapped harness. Devices not on the list are intercepted and given a mock factory instead.

Return type typing.Callable[[str], tango.DeviceProxy]

Returns a factory that putatively provides device connections, but might actually provide mocks.

class SatDeviceInfo(*path*, *package*, *devices=None*)
Data structure class that loads and holds information about devices.

It can provide that information in the format required by tango.test_context.MultiDeviceTestContext.

__init__(*path*, *package*, *devices=None*)
Create a new instance.

Parameters

- **path** – the path to the configuration file that contains information about all available devices.
- **package** – name of the package from which to draw classes
- **devices** – option specification of devices. If not provided, then devices can be added via the *include_device()* method.

as_mdtc_device_info()
Return this device info in a format required by MultiDeviceTestContext.

Returns device info in a format required by tango.test_context.MultiDeviceTestContext.

property fqdn_map: dict[str, str]
Return a dictionary that maps device names onto FQDNs.

Returns a mapping from device names to FQDNs

property fqdns: Iterable[str]
Return a list of device fqdns.

Return type typing.Iterable[str]

Returns a list of device FQDNs

include_device(*name*, *proxy*, *patch=None*)
Include a device in this specification.

Parameters

- **name** – the name of the device to be included. The source data must contain configuration information for a device listed under this name
- **proxy** – the proxy class to use to access the device.
- **patch** – an optional device class with which to patch the named device

Raises ValueError – if the named device does not exist in the source configuration data

property proxy_map: dict[str, type[SatDeviceProxy]]
Return a map from FQDN to proxy type.

Returns a map from FQDN to proxy type

class StartingStateTangoHarness(*harness*, *bypass_cache=True*, *check_ready=True*, *set_test_mode=True*, **args*, ***kwargs*)

A test harness for testing Tango devices.

It provides for certain actions and checks that ensure that devices are in a desired initial state prior to testing.

Specifically, it can:

- Tell devices to bypass their attribute cache, so that written values can be read back immediately
- Check that devices have completed initialisation and transitioned out of the INIT state
- Set device testMode to TestMode.TEST

__init__(*harness*, *bypass_cache=True*, *check_ready=True*, *set_test_mode=True*, **args*, ***kwargs*)

Initialise a new instance.

Parameters

- **harness** (*ska_sat_lmc.testing.tango_harness.TangoHarness*) – the wrapped harness
- **bypass_cache** (bool) – whether to tell each device to bypass its attribute cache so that written attribute values can be read back again immediately
- **check_ready** (bool) – whether to check whether each device has completed initialisation and transitioned out of INIT state before allowing tests to be run.
- **set_test_mode** (bool) – whether to set the device into test mode before allowing tests to be run.
- **args** (typing.Any) – additional positional arguments
- **kwargs** (typing.Any) – additional keyword arguments

class TangoHarness(**args*, ***kwargs*)

Abstract base class for Tango test harnesses.

This does very little, because it needs to support both harnesses that directly interact with Tango, and wrapper harnesses that add functionality to another harness.

The one really important thing it does do, is ensure that *ska_sat_lmc.device_proxy.SatDeviceProxy* uses this harness's `connection factory` to make connections.

__init__(**args*, ***kwargs*)

Initialise a new instance.

Parameters

- **args** (typing.Any) – additional positional arguments
- **kwargs** (typing.Any) – additional keyword arguments

property connection_factory: Callable[[str], tango.DeviceProxy]

Establish connections to devices with this factory.

Raises NotImplementedError – because this method is abstract

Return type typing.Callable[[str], tango.DeviceProxy]

property fqdns

Return FQDNs of devices in this harness.

Raises NotImplementedError – because this method is abstract

get_device(*fqdn*)
Create and return a proxy to the device at the given FQDN.

Parameters **fqdn** (str) – FQDN of the device for which a proxy is required

Raises NotImplementedError – because this method is abstract

Return type *ska_sat_lmc.device_proxy.SatDeviceProxy*

class TestContextTangoHarness(*device_info*, *logger*, *process=False*, **args*, ***kwargs*)
A test harness for testing SatLMC Tango devices in a lightweight test context.

It stands up a tango.test_context.MultiDeviceTestContext with the specified devices.

__init__(*device_info*, *logger*, *process=False*, **args*, ***kwargs*)
Initialise a new instance.

Parameters

- **device_info** (typing.Optional[*ska_sat_lmc.testing.tango_harness.SatDeviceInfo*]) – object that makes device info available
- **logger** (logging.Logger) – a logger for the harness
- **process** (bool) – whether to run the test context in a separate process or not
- **args** (typing.Any) – additional positional arguments
- **kwargs** (typing.Any) – additional keyword arguments

property connection_factory: Callable[[str], tango.DeviceProxy]
Establish connections to devices with this factory.

This class uses tango.DeviceProxy but patches it to use the long-form FQDN, as a workaround to an issue with tango.test_context.MultiDeviceTestContext.

Return type typing.Callable[[str], tango.DeviceProxy]

Returns a DeviceProxy for use in establishing connections.

1.5.2 Testing subpackage

This subpackage contains modules for test mocking in the SKA SatLMC tests.

class MockCallable(*return_value=None*, *called_timeout=5.0*, *not_called_timeout=1.0*)
This class implements a mock callable.

It is useful for when you want to assert that a callable is called, but the callback is called asynchronously, so that you might have to wait a short time for the call to occur.

If you use a regular mock for the callback, your tests will end up littered with sleeps:

```
antenna_apiu_proxy.start_communicating()
communication_status_changed_callback.assert_called_once_with(
    CommunicationStatus.NOT_ESTABLISHED
)
time.sleep(0.1)
communication_status_changed_callback.assert_called_once_with(
    CommunicationStatus.ESTABLISHED
)
```

These sleeps waste time, slow down the tests, and they are difficult to tune: maybe you only need to sleep 0.1 seconds on your development machine, but what if the CI pipeline deploys the tests to an environment that needs 0.2 seconds for this?

This class solves that by putting each call to the callback onto a queue. Then, each time we assert that a callback was called, we get a call from the queue, waiting if necessary for the call to arrive, but with a timeout:

```
antenna_apiu_proxy.start_communicating()
communication_status_changed_callback.assert_next_call(
    CommunicationStatus.NOT_ESTABLISHED
)
communication_status_changed_callback.assert_next_call(
    CommunicationStatus.ESTABLISHED
)
```

__init__(*return_value=None*, *called_timeout=5.0*, *not_called_timeout=1.0*)
Initialise a new instance.

Parameters

- **return_value** (typing.Optional[typing.Any]) – what to return when called
- **called_timeout** (float) – how long to wait for a call to occur when we are expecting one. It makes sense to wait a long time for the expected call, as it will generally arrive much much sooner anyhow, and failure for the call to arrive in time will cause the assertion to fail. The default is 5 seconds.
- **not_called_timeout** (float) – how long to wait for a callback when we are *not* expecting one. Since we need to wait the full timeout period in order to determine that a callback has not arrived, asserting that a call has not been made can severely slow down your tests. By keeping this timeout quite short, we can speed up our tests, at the risk of prematurely passing an assertion. The default is 0.5

assert_last_call(**args*, ***kwargs*)
Assert the arguments of the last call to this mock callback.

The "last" call is the last call before an attempt to get the next event times out.

This is useful for situations where we know a device may call a callback several time, and we don't care too much about the exact order of calls, but we do know what the final call should be.

Parameters

- **args** (typing.Any) – positional args that the call is asserted to have
- **kwargs** (typing.Any) – keyword args that the call is asserted to have

Raises AssertionError – if the callback has not been called.

Return type None

assert_next_call(**args*, ***kwargs*)
Assert the arguments of the next call to this mock callback.

If the call has not been made, this method will wait up to the specified timeout for a call to arrive.

Parameters

- **args** (typing.Any) – positional args that the call is asserted to have
- **kwargs** (typing.Any) – keyword args that the call is asserted to have

Raises AssertionError – if the callback has not been called.

Return type None

assert_not_called(*timeout=None*)

Assert that the callback still has not been called after the timeout period.

This is a slow method because it has to wait the full timeout period in order to determine that the call is not coming. An optional timeout parameter is provided for the situation where you are happy for the assertion to pass after a shorter wait time.

Parameters **timeout** (typing.Optional[float]) – optional timeout for the check. If not provided, the default is the class setting

Return type None

get_next_call()

Return the arguments of the next call to this mock callback.

This is useful for situations where you do not know exactly what the arguments of the next call will be, so you cannot use the *assert_next_call()* method. Instead you want to assert some specific properties on the arguments:

```
(args, kwargs) = mock_callback.get_next_call()
event_data = args[0].attr_value
assert event_data.name == "healthState"
assert event_data.value == HealthState.UNKNOWN
assert event_data.quality == tango.AttrQuality.ATTR_VALID
```

If the call has not been made, this method will wait up to the specified timeout for a call to arrive.

Raises **AssertionError** – if the callback has not been called

Return type typing.Tuple[typing.Sequence[typing.Any], typing.Sequence[typing.Any]]

Returns an (args, kwargs) tuple

class MockChangeEventCallback(*event_name*, *called_timeout=5.0*, *not_called_timeout=0.5*)

This class implements a mock change event callback.

It is a special case of a *MockCallable* where the callable expects to be called with event_name, event_value and event_quality arguments (which is how DeviceProxy calls its change event callbacks).

__init__(*event_name*, *called_timeout=5.0*, *not_called_timeout=0.5*)

Initialise a new instance.

Parameters

- **event_name** (str) – the name of the event for which this callable is a callback
- **called_timeout** (float) – how long to wait for a call to occur when we are expecting one. It makes sense to wait a long time for the expected call, as it will generally arrive much much sooner anyhow, and failure for the call to arrive in time will cause the assertion to fail. The default is 5 seconds.
- **not_called_timeout** (float) – how long to wait for a callback when we are *not* expecting one. Since we need to wait the full timeout period in order to determine that a callback has not arrived, asserting that a call has not been made can severely slow down your tests. By keeping this timeout quite short, we can speed up our tests, at the risk of prematurely passing an assertion. The default is 0.5

assert_last_change_event(*value*, *quality=tango.AttrQuality.ATTR_VALID*)

Assert the arguments of the last call to this mock callback.

The "last" call is the last call before an attempt to get the next event times out.

This is useful for situations where we know a device may fire several events, and we don't know or care about the exact order of events, but we do know what the final event should be. For example, when we tell a Controller to turn on, it has to turn many devices on, which have to turn many devices on, etc. With so m

Parameters

- **value** (typing.Any) – the asserted value of the change event
- **quality** (tango.AttrQuality) – the asserted quality of the change event. This is optional, with a default of ATTR_VALID.

Raises **AssertionError** – if the callback has not been called.

Return type None

assert_next_change_event(*value*, *quality=tango.AttrQuality.ATTR_VALID*)

Assert the arguments of the next call to this mock callback.

If the call has not been made, this method will wait up to the specified timeout for a call to arrive.

Parameters

- **value** (typing.Any) – the asserted value of the change event
- **quality** (tango.AttrQuality) – the asserted quality of the change event. This is optional, with a default of ATTR_VALID.

Raises **AssertionError** – if the callback has not been called.

Return type None

class MockDeviceBuilder(*from_factory=<class 'unittest.mock.Mock'>*)

This module implements a mock builder for tango devices.

__init__(*from_factory=<class 'unittest.mock.Mock'>*)

Create a new instance.

Parameters **from_factory** – an optional factory from which to draw the original mock

add_attribute(*name*, *value*)

Tell this builder to build mocks with a given attribute.

TODO: distinguish between read-only and read-write attributes

Parameters

- **name** (str) – name of the attribute
- **value** (typing.Any) – the value of the attribute

Return type None

add_command(*name*, *return_value*)

Tell this builder to build mocks with a specified command.

And that the command returns the provided value.

Parameters

- **name** (str) – name of the command
- **return_value** (typing.Any) – what the command should return

Return type None

add_result_command(*name*, *result_code*, *status='Mock information-only message'*)
Tell this builder to build mocks with a specified command.

And that the command returns (ResultCode, [message, message_uid]) or (ResultCode, message) tuples as required.

Parameters

- **name** (str) – the name of the command
- **result_code** (ska_tango_base.commands.ResultCode) – the ska_tango_base.commands.ResultCode that the command should return
- **status** (str) – an information-only message for the command to return

Return type None

set_state(*state*)
Tell this builder to build mocks with the state set as specified.

Parameters **state** (tango.DevState) – the state of the mock

Return type None

Mock callable

This module implements infrastructure for mocking callbacks and other callables.

class MockCallable(*return_value=None*, *called_timeout=5.0*, *not_called_timeout=1.0*)
This class implements a mock callable.

It is useful for when you want to assert that a callable is called, but the callback is called asynchronously, so that you might have to wait a short time for the call to occur.

If you use a regular mock for the callback, your tests will end up littered with sleeps:

```
antenna_apiu_proxy.start_communicating()
communication_status_changed_callback.assert_called_once_with(
    CommunicationStatus.NOT_ESTABLISHED
)
time.sleep(0.1)
communication_status_changed_callback.assert_called_once_with(
    CommunicationStatus.ESTABLISHED
)
```

These sleeps waste time, slow down the tests, and they are difficult to tune: maybe you only need to sleep 0.1 seconds on your development machine, but what if the CI pipeline deploys the tests to an environment that needs 0.2 seconds for this?

This class solves that by putting each call to the callback onto a queue. Then, each time we assert that a callback was called, we get a call from the queue, waiting if necessary for the call to arrive, but with a timeout:

```
antenna_apiu_proxy.start_communicating()
communication_status_changed_callback.assert_next_call(
    CommunicationStatus.NOT_ESTABLISHED
)
communication_status_changed_callback.assert_next_call(
    CommunicationStatus.ESTABLISHED
)
```

__init__(*return_value=None*, *called_timeout=5.0*, *not_called_timeout=1.0*)
Initialise a new instance.

Parameters

- **return_value** (typing.Optional[typing.Any]) – what to return when called
- **called_timeout** (float) – how long to wait for a call to occur when we are expecting one. It makes sense to wait a long time for the expected call, as it will generally arrive much much sooner anyhow, and failure for the call to arrive in time will cause the assertion to fail. The default is 5 seconds.
- **not_called_timeout** (float) – how long to wait for a callback when we are *not* expecting one. Since we need to wait the full timeout period in order to determine that a callback has not arrived, asserting that a call has not been made can severely slow down your tests. By keeping this timeout quite short, we can speed up our tests, at the risk of prematurely passing an assertion. The default is 0.5

assert_last_call(**args*, ***kwargs*)
Assert the arguments of the last call to this mock callback.

The "last" call is the last call before an attempt to get the next event times out.

This is useful for situations where we know a device may call a callback several time, and we don't care too much about the exact order of calls, but we do know what the final call should be.

Parameters

- **args** (typing.Any) – positional args that the call is asserted to have
- **kwargs** (typing.Any) – keyword args that the call is asserted to have

Raises AssertionError – if the callback has not been called.

Return type None

assert_next_call(**args*, ***kwargs*)
Assert the arguments of the next call to this mock callback.

If the call has not been made, this method will wait up to the specified timeout for a call to arrive.

Parameters

- **args** (typing.Any) – positional args that the call is asserted to have
- **kwargs** (typing.Any) – keyword args that the call is asserted to have

Raises AssertionError – if the callback has not been called.

Return type None

assert_not_called(*timeout=None*)
Assert that the callback still has not been called after the timeout period.

This is a slow method because it has to wait the full timeout period in order to determine that the call is not coming. An optional timeout parameter is provided for the situation where you are happy for the assertion to pass after a shorter wait time.

Parameters **timeout** (typing.Optional[float]) – optional timeout for the check. If not provided, the default is the class setting

Return type None

get_next_call()
Return the arguments of the next call to this mock callback.

This is useful for situations where you do not know exactly what the arguments of the next call will be, so you cannot use the *assert_next_call()* method. Instead you want to assert some specific properties on the arguments:

```
(args, kwargs) = mock_callback.get_next_call()
event_data = args[0].attr_value
assert event_data.name == "healthState"
assert event_data.value == HealthState.UNKNOWN
assert event_data.quality == tango.AttrQuality.ATTR_VALID
```

If the call has not been made, this method will wait up to the specified timeout for a call to arrive.

Raises AssertionError – if the callback has not been called

Return type typing.Tuple[typing.Sequence[typing.Any], typing.Sequence[typing.Any]]

Returns an (args, kwargs) tuple

class MockChangeEventCallback(*event_name*, *called_timeout=5.0*, *not_called_timeout=0.5*)

This class implements a mock change event callback.

It is a special case of a *MockCallable* where the callable expects to be called with event_name, event_value and event_quality arguments (which is how DeviceProxy calls its change event callbacks).

__init__(*event_name*, *called_timeout=5.0*, *not_called_timeout=0.5*)

Initialise a new instance.

Parameters

- **event_name** (str) – the name of the event for which this callable is a callback
- **called_timeout** (float) – how long to wait for a call to occur when we are expecting one. It makes sense to wait a long time for the expected call, as it will generally arrive much much sooner anyhow, and failure for the call to arrive in time will cause the assertion to fail. The default is 5 seconds.
- **not_called_timeout** (float) – how long to wait for a callback when we are *not* expecting one. Since we need to wait the full timeout period in order to determine that a callback has not arrived, asserting that a call has not been made can severely slow down your tests. By keeping this timeout quite short, we can speed up our tests, at the risk of prematurely passing an assertion. The default is 0.5

assert_last_change_event(*value*, *quality=tango.AttrQuality.ATTR_VALID*)

Assert the arguments of the last call to this mock callback.

The "last" call is the last call before an attempt to get the next event times out.

This is useful for situations where we know a device may fire several events, and we don't know or care about the exact order of events, but we do know what the final event should be. For example, when we tell a Controller to turn on, it has to turn many devices on, which have to turn many devices on, etc. With so m

Parameters

- **value** (typing.Any) – the asserted value of the change event
- **quality** (tango.AttrQuality) – the asserted quality of the change event. This is optional, with a default of ATTR_VALID.

Raises AssertionError – if the callback has not been called.

Return type None

assert_next_change_event(*value*, *quality=tango.AttrQuality.ATTR_VALID*)
Assert the arguments of the next call to this mock callback.

If the call has not been made, this method will wait up to the specified timeout for a call to arrive.

Parameters

- **value** (typing.Any) – the asserted value of the change event
- **quality** (tango.AttrQuality) – the asserted quality of the change event. This is optional, with a default of ATTR_VALID.

Raises AssertionError – if the callback has not been called.

Return type None

Mock device

This module implements infrastructure for mocking tango devices.

class MockDeviceBuilder(*from_factory=<class 'unittest.mock.Mock'>*)
This module implements a mock builder for tango devices.

__init__(*from_factory=<class 'unittest.mock.Mock'>*)
Create a new instance.

Parameters **from_factory** – an optional factory from which to draw the original mock

add_attribute(*name*, *value*)
Tell this builder to build mocks with a given attribute.

TODO: distinguish between read-only and read-write attributes

Parameters

- **name** (str) – name of the attribute
- **value** (typing.Any) – the value of the attribute

Return type None

add_command(*name*, *return_value*)
Tell this builder to build mocks with a specified command.

And that the command returns the provided value.

Parameters

- **name** (str) – name of the command
- **return_value** (typing.Any) – what the command should return

Return type None

add_result_command(*name*, *result_code*, *status='Mock information-only message'*)
Tell this builder to build mocks with a specified command.

And that the command returns (ResultCode, [message, message_uid]) or (ResultCode, message) tuples as required.

Parameters

- **name** (str) – the name of the command
- **result_code** (ska_tango_base.commands.ResultCode) – the ska_tango_base.commands.ResultCode that the command should return

- **status** (str) – an information-only message for the command to return

Return type None

set_state(*state*)
Tell this builder to build mocks with the state set as specified.

Parameters **state** (tango.DevState) – the state of the mock

Return type None

1.6 Comms

This module implements the Socket communication.

class TcpSocket(*host=None*, *port=None*, *eol=b'\\n'*, *timeout=30.0*, *logger=None*)
Socket communication class to access raw socket layer using TCP protocol.

Example from comms import TcpSocket sock = TcpSocket("tcp://127.0.0.1:45678")

class UdpSocket(*host=None*, *port=None*, *eol=b'\\n'*, *timeout=30.0*, *logger=None*)
Socket communication class to access raw socket layer using UDP protocol.

Example from comms import UdpSocket sock = UdpSocket("udp://127.0.0.1:45678")

parse_url(*url*)
Parse the url string into component host & port.

Parameters **url** – an encoded string containing host, port and socket type

Raises ValueError – invalid URL specified

Returns the decoded host & port

1.7 Device Proxy

This module implements a base device proxy for Sat LMC devices.

class SatDeviceProxy(*fqdn*, *logger*, *connect=True*, *connection_factory=None*, *pass_through=True*)
This class implements a base device proxy for Sat LMC devices.

At present it supports:

- deferred connection: we can create the proxy without immediately trying to connect to the proxied device.
- a :py:meth:`connect` method, for establishing that connection later
- a :py:meth:`check_initialised` method, for checking that / waiting until the proxied device has transitioned out of INIT state.
- Ability to subscribe to change events via the :py:meth:`add_change_event_callback` method.

__init__(*fqdn*, *logger*, *connect=True*, *connection_factory=None*, *pass_through=True*)
Create a new instance.

Parameters

- **fqdn** (str) – fqdn of the device to be proxied
- **logger** (logging.Logger) – a logger for this proxy to use

- **connection_factory** (typing.Optional[typing.Callable[[str], tango.DeviceProxy]]) – how we obtain a connection to the device we are proxying. By default this is tango.DeviceProxy, but occasionally this needs to be changed. For example, when testing against a tango.test_context.MultiDeviceTestContext, we obtain connections to the devices under test via `test_context.get_device(fqdn)`.
- **connect** (bool) – whether to connect immediately to the device. If False, then the device may be connected later by calling the *connect()* method.
- **pass_through** (bool) – whether to pass unrecognised attribute accesses through to the underlying connection. Defaults to `True` but this will likely change in future once our proxies are more mature.

add_change_event_callback(*attribute_name*, *callback*, *stateless=True*)
Register a callback for change events being pushed by the device.

Parameters

- **attribute_name** (str) – the name of the attribute for which change events are subscribed.
- **callback** (typing.Callable[[str, typing.Any, tango.AttrQuality], None]) – the function to be called when a change event arrives.
- **stateless** (bool) – whether to use Tango's stateless subscription feature

Return type None

check_initialised(*max_time=120.0*)
Check that the device has completed initialisation.

That is, check that the device is no longer in state INIT.

Parameters **max_time** (float) – the (optional) maximum time, in seconds, to wait for the device to complete initialisation. The default is 120.0 i.e. two minutes. If set to 0 or None, the device is checked once and the call returns immediately.

Return type bool

Returns whether the device is initialised yet

connect(*max_time=120.0*)
Establish a connection to the device that we want to proxy.

Parameters **max_time** (float) – the maximum time, in seconds, to wait for a connection to be established. The default is 120 i.e. two minutes. If set to 0 or None, a single connection attempt is made, and the call returns immediately.

Return type None

classmethod set_default_connection_factory(*connection_factory*)
Set the default connection factory for this class.

This is super useful for unit testing: we can mock out tango.DeviceProxy altogether, by simply setting this class's default connection factory to a mock factory.

Parameters **connection_factory** (typing.Callable[[str], tango.DeviceProxy]) – default factory to use to establish a connection to the device

Return type None

1.8 Health

This module implements infrastructure for health management.

class HealthModel(*health_changed_callback*)

A simple health model the supports.

- HealthState.UNKNOWN – when communication with the component is not established.
- HealthState.FAILED – when the component has faulted
- HealthState.OK – when neither of the above conditions holds.

This health model does not support HealthState.DEGRADED. It is up to subclasses to implement support for DEGRADED if required.

__init__(*health_changed_callback*)

Initialise a new instance.

Parameters **health_changed_callback** (typing.Callable[[ska_tango_base.control_model.HealthState], None]) – callback to be called whenever there is a change to this this health model's evaluated health state.

component_fault(*faulty*)

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 **faulty** (bool) – whether the component has faulted or not

Return type None

evaluate_health()

Re-evaluate the health state.

This method contains the logic for evaluating the health. It is this method that should be extended by subclasses in order to define how health is evaluated by their particular device.

Return type ska_tango_base.control_model.HealthState

Returns the new health state.

property health_state: ska_tango_base.control_model.HealthState

Return the health state.

Return type ska_tango_base.control_model.HealthState

Returns the health state.

is_communicating(*communicating*)

Handle change in communication with the component.

Parameters **communicating** (bool) – whether communications with the component is established.

Return type None

update_health()

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.

Return type None

1.9 Release

Release information for SKA SAT LMC Python Package.

get_release_info(*clsname=None*)
Return a formatted release info string.

Parameters **clsname** (*str*) – optional name of class to add to the info

Return type str

Returns str

1.10 Utils

Module for utils.

class ThreadsafeCheckingMeta(*name: str*, *bases: tuple[type]*, *attrs: dict*)
Metaclass that checks for methods being run by multiple concurrent threads.

call_with_json(*func*, ***kwargs*)
Call a command with a json string.

Allows the calling of a command that accepts a JSON string as input, with the actual unserialised parameters.

For example, suppose you need to use *Allocate(resources)* command to tell a controller device to allocate certain stations and tiles to a subarray. *Allocate* accepts a single JSON string argument. Instead of

Example:

```
parameters={"id": id, "stations": stations, "tiles": tiles}
json_string=json.dumps(parameters)
controller.Allocate(json_string)
```

save yourself the trouble and

Example:

```
call_with_json(controller.Allocate, id=id, stations=stations, tiles=tiles)
```

Parameters

- **func** (typing.Callable) – the function handle to call
- **kwargs** (typing.Any) – parameters to be jsonified and passed to func

Return type typing.Any

Returns the return value of func

class json_input(*schema_path=None*)
Parse and validate json string input.

Method decorator that parses and validates JSON input into a python dictionary, which is then passed to the method as kwargs. The wrapped method is thus called with a JSON string, but can be implemented as if it had been passed a sequence of named arguments.

If the string cannot be parsed as JSON, an exception is raised.

For example, conceptually, Controller.Allocate() takes as arguments a subarray id, an array of stations, and an array of tiles. In practice, however, these arguments are encoded into a JSON string. Implement the function with its conceptual parameters, then wrap it in this decorator:

Example:

```
@json_input
def Controller.Allocate(id, stations, tiles):
```

The decorator will provide the JSON interface and handle the decoding for you.

__init__(*schema_path=None*)
Initialise a callable json_input object.

To function as a device method generator.

Parameters **schema_path** (typing.Optional[str]) – an optional path to a schema against which the JSON should be validated. Not working at the moment, so leave it None.

threadsafe(*func*)
Use this method as a decorator for marking a method as threadsafe.

This tells the ThreadsafeCheckingMeta metaclass that it is okay for the decorated method to have more than one thread in it at a time. The metaclass will still raise an exception if the *same* thread enters the method multiple times, because re-entry is a common cause of deadlock.

Parameters **func** (typing.Callable) – the method to be marked as threadsafe

Return type typing.Callable

Returns the method, marked as threadsafe

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

ska_sat_lmc.comms, 65
ska_sat_lmc.component, 46
ska_sat_lmc.component.component_manager, 49
ska_sat_lmc.component.device_component_manager, 51
ska_sat_lmc.component.util, 52
ska_sat_lmc.controller, 3
ska_sat_lmc.controller.controller_component_manager, 5
ska_sat_lmc.controller.controller_device, 6
ska_sat_lmc.controller.controller_health_model, 7
ska_sat_lmc.device_proxy, 65
ska_sat_lmc.health, 67
ska_sat_lmc.maser, 8
ska_sat_lmc.maser.maser_component_manager, 20
ska_sat_lmc.maser.maser_device, 22
ska_sat_lmc.maser.maser_health_model, 29
ska_sat_lmc.phasemicrostepper, 29
ska_sat_lmc.phasemicrostepper.phasemicrostepper_component_manager, 37
ska_sat_lmc.phasemicrostepper.phasemicrostepper_device, 38
ska_sat_lmc.phasemicrostepper.phasemicrostepper_driver, 43
ska_sat_lmc.phasemicrostepper.phasemicrostepper_health_model, 46
ska_sat_lmc.release, 68
ska_sat_lmc.testing, 53
ska_sat_lmc.testing.mock, 57
ska_sat_lmc.testing.mock.mock_callable, 61
ska_sat_lmc.testing.mock.mock_device, 64
ska_sat_lmc.testing.tango_harness, 53
ska_sat_lmc.utils, 68

INDEX

Symbols

__init__() (*BaseTangoHarness method*), 53
__init__() (*ClientProxyTangoHarness method*), 54
__init__() (*ControllerComponentManager method*), 3, 5
__init__() (*ControllerHealthModel method*), 4, 7
__init__() (*DeviceComponentManager method*), 46, 51
__init__() (*HealthModel method*), 67
__init__() (*MaserComponentManager method*), 8, 20
__init__() (*MaserDriver method*), 8
__init__() (*MockCallable method*), 58, 61
__init__() (*MockChangeEventCallback method*), 59, 63
__init__() (*MockDeviceBuilder method*), 60, 64
__init__() (*MockingTangoHarness method*), 54
__init__() (*PhaseMicroStepperComponentManager method*), 29, 37
__init__() (*PhaseMicroStepperDriver method*), 30, 43
__init__() (*SatComponentManager method*), 47, 50
__init__() (*SatDeviceInfo method*), 55
__init__() (*SatDeviceProxy method*), 65
__init__() (*StartingStateTangoHarness method*), 56
__init__() (*TangoHarness method*), 53, 56
__init__() (*TestContextTangoHarness method*), 57
__init__() (*json_input method*), 69

A

add_attribute() (*MockDeviceBuilder method*), 60, 64
add_change_event_callback() (*SatDeviceProxy method*), 66
add_command() (*MockDeviceBuilder method*), 60, 64
add_result_command() (*MockDeviceBuilder method*), 60, 64
Advance10M() (*SatPhaseMicroStepper method*), 32, 38
advance_10megahertz() (*PhaseMicroStepperDriver method*), 30, 44
advance_phase() (*PhaseMicroStepperDriver method*), 30, 44
advance_pps() (*PhaseMicroStepperDriver method*), 30, 44
AdvanceAllPhase() (*SatPhaseMicroStepper method*), 33, 39
AdvancePPS() (*SatPhaseMicroStepper method*), 33, 39
ambient_temperature (*MaserDriver property*), 9
ambient_temperature() (*SatMaser method*), 14, 23
amplitude_405khz_voltage (*MaserDriver property*), 9
amplitude_405khz_voltage() (*SatMaser method*), 14, 24
as_mdtc_device_info() (*SatDeviceInfo method*), 55
assert_last_call() (*MockCallable method*), 58, 62
assert_last_change_event() (*MockChangeEventCallback method*), 59, 63
assert_next_call() (*MockCallable method*), 58, 62
assert_next_change_event() (*MockChangeEventCallback method*), 60, 63
assert_not_called() (*MockCallable method*), 59, 62

B

BaseTangoHarness (*class in ska_sat_lmc.testing.tango_harness*), 53
battery_current_a (*MaserDriver property*), 9
battery_current_a() (*SatMaser method*), 14, 24
battery_current_b (*MaserDriver property*), 9
battery_current_b() (*SatMaser method*), 15, 24
battery_voltage_a (*MaserDriver property*), 9
battery_voltage_a() (*SatMaser method*), 15, 24
battery_voltage_b (*MaserDriver property*), 9
battery_voltage_b() (*SatMaser method*), 15, 24
boxes_current (*MaserDriver property*), 9
boxes_current() (*SatMaser method*), 15, 24
boxes_temperature (*MaserDriver property*), 9
boxes_temperature() (*SatMaser method*), 15, 24

C

call_with_json() (*in module ska_sat_lmc.utils*), 68
cfield_voltage (*MaserDriver property*), 10
cfield_voltage() (*SatMaser method*), 15, 24
check_communicating() (*in module ska_sat_lmc.component*), 49
check_communicating() (*in module ska_sat_lmc.component.util*), 52

check_initialised() (*SatDeviceProxy method*), 66
ClientProxyTangoHarness (*class in ska_sat_lmc.testing.tango_harness*), 54
communication_status (*SatComponentManager property*), 48, 50
CommunicationStatus (*class in ska_sat_lmc.component*), 46
CommunicationStatus (*class in ska_sat_lmc.component.component_manager*), 49
component_fault() (*HealthModel method*), 67
component_fault_changed() (*SatComponentManager method*), 48, 50
connect() (*SatDeviceProxy method*), 66
connection_factory (*BaseTangoHarness property*), 54
connection_factory (*MockingTangoHarness property*), 55
connection_factory (*TangoHarness property*), 53, 56
connection_factory (*TestContextTangoHarness property*), 57
ControllerComponentManager (*class in ska_sat_lmc.controller*), 3
ControllerComponentManager (*class in ska_sat_lmc.controller.controller_component_manager*), 5
ControllerHealthModel (*class in ska_sat_lmc.controller*), 4
ControllerHealthModel (*class in ska_sat_lmc.controller.controller_health_model*), 7
create_component_manager() (*SatController method*), 4, 6
create_component_manager() (*SatMaser method*), 15, 24
create_component_manager() (*SatPhaseMicroStepper method*), 35, 41

D

DeviceComponentManager (*class in ska_sat_lmc.component*), 46
DeviceComponentManager (*class in ska_sat_lmc.component.device_component_manager*), 51
DISABLED (*CommunicationStatus attribute*), 46, 49
dissociator_current (*MaserDriver property*), 10
dissociator_current() (*SatMaser method*), 15, 24
dissociator_light (*MaserDriver property*), 10
dissociator_light() (*SatMaser method*), 15, 25
do() (*SatController.InitCommand method*), 4, 6
do() (*SatMaser.InitCommand method*), 13, 23
do() (*SatPhaseMicroStepper.Advance10MCommand method*), 32, 39
do() (*SatPhaseMicroStepper.AdvanceAllPhaseCommand method*), 33, 39
do() (*SatPhaseMicroStepper.AdvancePPSCommand method*), 33, 39
do() (*SatPhaseMicroStepper.GetOffsetFrequencyCommand method*), 33, 40
do() (*SatPhaseMicroStepper.InitCommand method*), 34, 40
do() (*SatPhaseMicroStepper.SetIPAddressCommand method*), 34, 40
do() (*SatPhaseMicroStepper.SetOffsetFrequencyCommand method*), 34, 40
do() (*SatPhaseMicroStepper.SetUDPAddressCommand method*), 34, 40
do() (*SatPhaseMicroStepper.SyncPPSCommand method*), 35, 41

E

ESTABLISHED (*CommunicationStatus attribute*), 46, 49
evaluate_health() (*ControllerHealthModel method*), 4, 7
evaluate_health() (*HealthModel method*), 67
external_bottom_heater_voltage (*MaserDriver property*), 10
external_bottom_heater_voltage() (*SatMaser method*), 15, 25
external_high_current_value (*MaserDriver property*), 10
external_high_current_value() (*SatMaser method*), 16, 25
external_high_voltage_value (*MaserDriver property*), 10
external_high_voltage_value() (*SatMaser method*), 16, 25
external_side_heater_voltage (*MaserDriver property*), 10
external_side_heater_voltage() (*SatMaser method*), 16, 25

F

faulty (*SatComponentManager property*), 48, 50
first_internal_oscillator (*PhaseMicroStepperDriver property*), 31, 44
first_internal_oscillator() (*SatPhaseMicroStepper method*), 35, 41
fqdn_map (*SatDeviceInfo property*), 55
fqdns (*BaseTangoHarness property*), 54
fqdns (*SatDeviceInfo property*), 55
fqdns (*TangoHarness property*), 53, 56

G

get_device() (*BaseTangoHarness method*), 54
get_device() (*ClientProxyTangoHarness method*), 54
get_device() (*TangoHarness method*), 53, 56
get_next_call() (*MockCallable method*), 59, 62
get_offset_frequency() (*PhaseMicroStepperDriver method*), 31, 44
get_release_info() (*in module ska_sat_lmc.release*), 68
GetOffsetFrequency() (*SatPhaseMicroStepper method*), 33, 39
GetVersionInfo() (*SatMaser method*), 13, 23

H

health (*DeviceComponentManager property*), 47, 52
health_changed() (*SatController method*), 4, 6
health_changed() (*SatMaser method*), 16, 25
health_changed() (*SatPhaseMicroStepper method*), 35, 41
health_state (*HealthModel property*), 67
HealthModel (*class in ska_sat_lmc.health*), 67
hydrogen_pressure_measured (*MaserDriver property*), 10
hydrogen_pressure_measured() (*SatMaser method*), 16, 25
hydrogen_pressure_setting (*MaserDriver property*), 10
hydrogen_pressure_setting() (*SatMaser method*), 16, 25
hydrogen_storage_heater_voltage (*MaserDriver property*), 10
hydrogen_storage_heater_voltage() (*SatMaser method*), 16, 25
hydrogen_storage_pressure (*MaserDriver property*), 11
hydrogen_storage_pressure() (*SatMaser method*), 16, 26

I

include_device() (*SatDeviceInfo method*), 55
init_command_objects() (*SatController method*), 5, 6
init_command_objects() (*SatMaser method*), 16, 26
init_command_objects() (*SatPhaseMicroStepper method*), 35, 41
init_device() (*SatController method*), 5, 6
init_device() (*SatMaser method*), 17, 26
init_device() (*SatPhaseMicroStepper method*), 35, 42
internal_bottom_heater_voltage (*MaserDriver property*), 11
internal_bottom_heater_voltage() (*SatMaser method*), 17, 26
internal_high_current_value (*MaserDriver property*), 11
internal_high_current_value() (*SatMaser method*), 17, 26
internal_high_voltage_value (*MaserDriver property*), 11
internal_high_voltage_value() (*SatMaser method*), 17, 26
internal_side_heater_voltage (*MaserDriver property*), 11
internal_side_heater_voltage() (*SatMaser method*), 17, 26
internal_temperature (*PhaseMicroStepperDriver property*), 31, 44
internal_temperature() (*SatPhaseMicroStepper method*), 36, 42
internal_top_heater_voltage (*MaserDriver property*), 11
internal_top_heater_voltage() (*SatMaser method*), 17, 26
is_Advance10M_allowed() (*SatPhaseMicroStepper method*), 36, 42
is_AdvanceAllPhase_allowed() (*SatPhaseMicroStepper method*), 36, 42
is_AdvancePPS_allowed() (*SatPhaseMicroStepper method*), 36, 42
is_attribute_allowed() (*SatMaser method*), 17, 26
is_attribute_allowed() (*SatPhaseMicroStepper method*), 36, 42
is_communicating (*SatComponentManager property*), 48, 51
is_communicating() (*HealthModel method*), 67
is_SetIPaddress_allowed() (*SatPhaseMicroStepper method*), 36, 42
is_SetUDPaddress_allowed() (*SatPhaseMicroStepper method*), 36, 42
is_SyncPPS_allowed() (*SatPhaseMicroStepper method*), 36, 42
isolator_heater_voltage (*MaserDriver property*), 11
isolator_heater_voltage() (*SatMaser method*), 17, 27

J

json_input (*class in ska_sat_lmc.utils*), 68

L

lock100mhz (*MaserDriver property*), 11
lock100mhz() (*SatMaser method*), 17, 27

M

main() (*in module ska_sat_lmc.controller.controller_device*), 6
main() (*in module ska_sat_lmc.maser.maser_device*), 28
main() (*in module ska_sat_lmc.phasemicrostepper.phasemicrostepper_dev*
43

maser_health_changed() (*ControllerHealthModel method*), 4, 7
MaserComponentManager (*class in ska_sat_lmc.maser*), 8
MaserComponentManager (*class in ska_sat_lmc.maser.maser_component_manager*), 20
MaserDriver (*class in ska_sat_lmc.maser*), 8
MaserHealthModel (*class in ska_sat_lmc.maser*), 13
MaserHealthModel (*class in ska_sat_lmc.maser.maser_health_model*), 29
MockCallable (*class in ska_sat_lmc.testing.mock*), 57
MockCallable (*class in ska_sat_lmc.testing.mock.mock_callable*), 61
MockChangeEventCallback (*class in ska_sat_lmc.testing.mock*), 59
MockChangeEventCallback (*class in ska_sat_lmc.testing.mock.mock_callable*), 63
MockDeviceBuilder (*class in ska_sat_lmc.testing.mock*), 60
MockDeviceBuilder (*class in ska_sat_lmc.testing.mock.mock_device*), 64
MockingTangoHarness (*class in ska_sat_lmc.testing.tango_harness*), 54
module
ska_sat_lmc.comms, 65
ska_sat_lmc.component, 46
ska_sat_lmc.component.component_manager, 49
ska_sat_lmc.component.device_component_manager, 51
ska_sat_lmc.component.util, 52
ska_sat_lmc.controller, 3
ska_sat_lmc.controller.controller_component_manager, 5
ska_sat_lmc.controller.controller_device, 6
ska_sat_lmc.controller.controller_health_model, 7
ska_sat_lmc.device_proxy, 65
ska_sat_lmc.health, 67
ska_sat_lmc.maser, 8
ska_sat_lmc.maser.maser_component_manager, 20
ska_sat_lmc.maser.maser_device, 22
ska_sat_lmc.maser.maser_health_model, 29
ska_sat_lmc.phasemicrostepper, 29
ska_sat_lmc.phasemicrostepper.phasemicrostepper_component_manager, 37
ska_sat_lmc.phasemicrostepper.phasemicrostepper_device, 38
ska_sat_lmc.phasemicrostepper.phasemicrostepper_driver, 43
ska_sat_lmc.phasemicrostepper.phasemicrostepper_health, 46
ska_sat_lmc.release, 68
ska_sat_lmc.testing, 53
ska_sat_lmc.testing.mock, 57
ska_sat_lmc.testing.mock.mock_callable, 61
ska_sat_lmc.testing.mock.mock_device, 64
ska_sat_lmc.testing.tango_harness, 53
ska_sat_lmc.utils, 68

N

negative15vdc (*MaserDriver property*), 11
negative15vdc() (*SatMaser method*), 18, 27
negative5vdc (*MaserDriver property*), 11
negative5vdc() (*SatMaser method*), 18, 27
NOT_ESTABLISHED (*CommunicationStatus attribute*), 46, 49

O

off() (*ControllerComponentManager method*), 3, 5
Off() (*SatMaser method*), 14, 23
on() (*ControllerComponentManager method*), 3, 5
On() (*SatMaser method*), 14, 23
oscillator_100mhz_voltage (*MaserDriver property*), 12
oscillator_100mhz_voltage() (*SatMaser method*), 18, 27
oscillator_voltage (*MaserDriver property*), 12
oscillator_voltage() (*SatMaser method*), 18, 27

P

parse_url() (*in module ska_sat_lmc.comms*), 65
phase_lock_loop_lockstatus (*MaserDriver property*), 12
phase_lock_loop_lockstatus() (*SatMaser method*), 18, 27
phase_thread_running() (*PhaseMicroStepperDriver method*), 31, 45
PhaseMicroStepperComponentManager (*class in ska_sat_lmc.phasemicrostepper*), 29
PhaseMicroStepperComponentManager (*class in ska_sat_lmc.phasemicrostepper.phasemicrostepper_component_manager*), 37
PhaseMicroStepperDriver (*class in ska_sat_lmc.phasemicrostepper*), 30
PhaseMicroStepperDriver (*class in ska_sat_lmc.phasemicrostepper.phasemicrostepper_driver*), 43
PhaseMicroStepperHealthModel (*class in ska_sat_lmc.phasemicrostepper*), 32

PhaseMicroStepperHealthModel (*class in ska_sat_lmc.phasemicrostepper.phasemicrostepper_health_model*), 46
pirani_heater_voltage (*MaserDriver property*), 12
pirani_heater_voltage() (*SatMaser method*), 18, 27
positive15vdc (*MaserDriver property*), 12
positive15vdc() (*SatMaser method*), 18, 27
positive18vdc (*MaserDriver property*), 12
positive18vdc() (*SatMaser method*), 18, 27
positive24vdc (*MaserDriver property*), 12
positive24vdc() (*SatMaser method*), 18, 28
positive5vdc (*MaserDriver property*), 12
positive5vdc() (*SatMaser method*), 18, 28
positive8vdc (*MaserDriver property*), 12
positive8vdc() (*SatMaser method*), 19, 28
proxy_map (*SatDeviceInfo property*), 55
purifier_current (*MaserDriver property*), 13
purifier_current() (*SatMaser method*), 19, 28

R

Reset() (*SatMaser method*), 14, 23

S

SatComponentManager (*class in ska_sat_lmc.component*), 47
SatComponentManager (*class in ska_sat_lmc.component.component_manager*), 50
SatController (*class in ska_sat_lmc.controller*), 4
SatController (*class in ska_sat_lmc.controller.controller_device*), 6
SatController.InitCommand (*class in ska_sat_lmc.controller*), 4
SatController.InitCommand (*class in ska_sat_lmc.controller.controller_device*), 6
SatDeviceInfo (*class in ska_sat_lmc.testing.tango_harness*), 55
SatDeviceProxy (*class in ska_sat_lmc.device_proxy*), 65
SatMaser (*class in ska_sat_lmc.maser*), 13
SatMaser (*class in ska_sat_lmc.maser.maser_device*), 22
SatMaser.InitCommand (*class in ska_sat_lmc.maser*), 13
SatMaser.InitCommand (*class in ska_sat_lmc.maser.maser_device*), 23
SatPhaseMicroStepper (*class in ska_sat_lmc.phasemicrostepper*), 32
SatPhaseMicroStepper (*class in ska_sat_lmc.phasemicrostepper.phasemicrostepper_device*), 38
SatPhaseMicroStepper.Advance10MCommand (*class in ska_sat_lmc.phasemicrostepper*), 32
SatPhaseMicroStepper.Advance10MCommand (*class in ska_sat_lmc.phasemicrostepper.phasemicrostepper_device*), 38
SatPhaseMicroStepper.AdvanceAllPhaseCommand (*class in ska_sat_lmc.phasemicrostepper*), 33
SatPhaseMicroStepper.AdvanceAllPhaseCommand (*class in ska_sat_lmc.phasemicrostepper.phasemicrostepper_device*), 39
SatPhaseMicroStepper.AdvancePPSCommand (*class in ska_sat_lmc.phasemicrostepper*), 33
SatPhaseMicroStepper.AdvancePPSCommand (*class in ska_sat_lmc.phasemicrostepper.phasemicrostepper_device*), 39
SatPhaseMicroStepper.GetOffsetFrequencyCommand (*class in ska_sat_lmc.phasemicrostepper*), 33
SatPhaseMicroStepper.GetOffsetFrequencyCommand (*class in ska_sat_lmc.phasemicrostepper.phasemicrostepper_device*), 39
SatPhaseMicroStepper.InitCommand (*class in ska_sat_lmc.phasemicrostepper*), 34
SatPhaseMicroStepper.InitCommand (*class in ska_sat_lmc.phasemicrostepper.phasemicrostepper_device*), 40
SatPhaseMicroStepper.SetIPAddressCommand (*class in ska_sat_lmc.phasemicrostepper*), 34
SatPhaseMicroStepper.SetIPAddressCommand (*class in ska_sat_lmc.phasemicrostepper.phasemicrostepper_device*), 40
SatPhaseMicroStepper.SetOffsetFrequencyCommand (*class in ska_sat_lmc.phasemicrostepper*), 34
SatPhaseMicroStepper.SetOffsetFrequencyCommand (*class in ska_sat_lmc.phasemicrostepper.phasemicrostepper_device*), 40
SatPhaseMicroStepper.SetUDPAddressCommand (*class in ska_sat_lmc.phasemicrostepper*), 34
SatPhaseMicroStepper.SetUDPAddressCommand (*class in ska_sat_lmc.phasemicrostepper.phasemicrostepper_device*), 40
SatPhaseMicroStepper.SyncPPSCommand (*class in ska_sat_lmc.phasemicrostepper*), 35
SatPhaseMicroStepper.SyncPPSCommand (*class in ska_sat_lmc.phasemicrostepper.phasemicrostepper_device*), 41
second_internal_oscillator (*PhaseMicroStepperDriver property*), 31, 45
second_internal_oscillator() (*SatPhaseMicroStepper method*), 36, 42
serial_number (*PhaseMicroStepperDriver property*), 31, 45
serial_number() (*SatPhaseMicroStepper method*), 37, 43
set_default_connection_factory() (*SatDeviceProxy class method*), 66
set_ip_address() (*PhaseMicroStepperDriver*

method), 31, 45
set_offset_frequency() (*PhaseMicroStepperDriver method*), 31, 45
set_state() (*MockDeviceBuilder method*), 61, 65
set_udp_address() (*PhaseMicroStepperDriver method*), 32, 45
SetIPaddress() (*SatPhaseMicroStepper method*), 34, 40
SetOffsetFrequency() (*SatPhaseMicroStepper method*), 34, 40
SetUDPaddress() (*SatPhaseMicroStepper method*), 35, 41
simulation_mode (*MaserComponentManager property*), 8, 20
simulation_mode (*PhaseMicroStepperComponentManager property*), 29, 37
simulationMode() (*SatMaser method*), 19, 28
simulationMode() (*SatPhaseMicroStepper method*), 37, 43
simulator_thread_running() (*PhaseMicroStepperDriver method*), 32, 45
ska_sat_lmc.comms
module, 65
ska_sat_lmc.component
module, 46
ska_sat_lmc.component.component_manager
module, 49
ska_sat_lmc.component.device_component_manager
module, 51
ska_sat_lmc.component.util
module, 52
ska_sat_lmc.controller
module, 3
ska_sat_lmc.controller.controller_component_manager
module, 5
ska_sat_lmc.controller.controller_device
module, 6
ska_sat_lmc.controller.controller_health_model
module, 7
ska_sat_lmc.device_proxy
module, 65
ska_sat_lmc.health
module, 67
ska_sat_lmc.maser
module, 8
ska_sat_lmc.maser.maser_component_manager
module, 20
ska_sat_lmc.maser.maser_device
module, 22
ska_sat_lmc.maser.maser_health_model
module, 29
ska_sat_lmc.phasemicrostepper
module, 29
ska_sat_lmc.phasemicrostepper.phasemicrostepper_component_manager
module, 37
ska_sat_lmc.phasemicrostepper.phasemicrostepper_device
module, 38
ska_sat_lmc.phasemicrostepper.phasemicrostepper_driver
module, 43
ska_sat_lmc.phasemicrostepper.phasemicrostepper_health_mod
module, 46
ska_sat_lmc.release
module, 68
ska_sat_lmc.testing
module, 53
ska_sat_lmc.testing.mock
module, 57
ska_sat_lmc.testing.mock.mock_callable
module, 61
ska_sat_lmc.testing.mock.mock_device
module, 64
ska_sat_lmc.testing.tango_harness
module, 53
ska_sat_lmc.utils
module, 68
Standby() (*SatMaser method*), 14, 23
start_communicating() (*ControllerComponentManager method*), 3, 5
start_communicating() (*DeviceComponentManager method*), 47, 52
start_communicating() (*MaserDriver method*), 13
start_communicating() (*PhaseMicroStepperDriver method*), 32, 45
start_communicating() (*SatComponentManager method*), 48, 51
StartingStateTangoHarness (*class in ska_sat_lmc.testing.tango_harness*), 56
stop_communicating() (*ControllerComponentManager method*), 3, 5
stop_communicating() (*DeviceComponentManager method*), 47, 52
stop_communicating() (*MaserDriver method*), 13
stop_communicating() (*PhaseMicroStepperDriver method*), 32, 45
stop_communicating() (*SatComponentManager method*), 48, 51
sync_pps() (*PhaseMicroStepperDriver method*), 32, 46
SyncPPS() (*SatPhaseMicroStepper method*), 35, 41

T

TangoHarness (*class in ska_sat_lmc.testing*), 53
TangoHarness (*class in ska_sat_lmc.testing.tango_harness*), 56
TcpSocket (*class in ska_sat_lmc.comms*), 65
test_mode (*MaserComponentManager property*), 8, 20
test_mode (*PhaseMicroStepperComponentManager property*), 30, 38

TestContextTangoHarness (*class in ska_sat_lmc.testing.tango_harness*), 57
testMode() (*SatMaser method*), 19, 28
testMode() (*SatPhaseMicroStepper method*), 37, 43
thermal_control_unit_heater_voltage (*MaserDriver property*), 13
thermal_control_unit_heater_voltage() (*SatMaser method*), 19, 28
threadsafe() (*in module ska_sat_lmc.utils*), 69
ThreadsafeCheckingMeta (*class in ska_sat_lmc.utils*), 68
tube_heater_voltage (*MaserDriver property*), 13
tube_heater_voltage() (*SatMaser method*), 19, 28

U

UdpSocket (*class in ska_sat_lmc.comms*), 65
update_communication_status() (*SatComponentManager method*), 48, 51
update_component_fault() (*SatComponentManager method*), 49, 51
update_health() (*HealthModel method*), 67

V

varactor_diode_voltage (*MaserDriver property*), 13
varactor_diode_voltage() (*SatMaser method*), 19, 28

W

wait_for_maser_thread_running() (*MaserDriver method*), 13
wait_for_simulator_thread_running() (*MaserDriver method*), 13