$\mathbf{py}_{dssinterface}$

Release 1.0.2

Jul 23, 2022

Contents

CHAPTER 1

Overview

docs	
tests	|requires|
package	

py-dss-interface is a Windows Python package providing access to OpenDSS direct dll version of OpenDSS - Version 9.2.0.1 (64-bit build); License Status: Open and Version 9.2.0.1 (32-bit build); License Status: Open.

- Free software: MIT license

1.1 Disclaimer

This Python Package is purely responsibility of Paulo Radatz and not his employer. Use this package at your own risk.

1.2 Installation

```
pip install py-dss-interface
```

1.3 Documentation

You can access the documentation through:

1 - The Read the Docs.

https://py_dss_interface.readthedocs.io/

2 - Another good resource is the OpenDSS_Direct_DLL.pdf doc created by Davis Montenegro. The package has been done based on this documentation.

https://sourceforge.net/p/electricdss/code/HEAD/tree/trunk/Version8/Distrib/Doc/OpenDSS_Direct_DLL.pdf

1.4 Thanks

I want to thank Ênio Viana and Rodolfo Pilar Londero for all their contribution to the new version of the tool.

CHAPTER 2

Installation

At the command line:

```
pip install py-dss-interface
```

CHAPTER 3

How to Use py-dss-interface Python package

3.1 py-dss-interface Python package

py-dss-interface is a Windows Python package providing access to OpenDSS direct dll.

3.2 Simple Usage

First import the Package

Creates an OpenDSS object

```
OpenDSS Started successfully!
OpenDSS Version 9.3.0.1 (64-bit build); License Status: Open
```

If you want to use your OpenDSS, you will need to pass the OpenDSS path as argument to the DSSDLL class, as can be seen below:

Creates a dss object with your OpenDSS

```
OpenDSS Started successfully!
OpenDSS Version 9.2.0.1 (64-bit build); License Status: Open
```

Select the DSS model

Compile

```
''
```

Solve - You can use the text interface as well: dss.text("solve")

```
0
```

Show Voltage Report

```
''
```

Get all buses voltages

```
[57499.9999611886, 33197.64035949601, -0.00013449617643016606, -66395.28088703856, -
↪57499.99982669239, 33197.64054049976, 2521.7954836507547, -0.11910264415748555, -
↪1245.983653171188, -2157.930119652756, -1260.782309001901, 2184.0080773974814, 2461.
↪932156820345, -76.36035906329018, -1284.5460767592406, -2134.6137221915546, -1162.
↪2387444138396, 2149.367176421046, 2447.622527822158, -105.33323048056116, -1292.
↪0799754959983, -2145.0014944867958, -1135.8268535116601, 2139.8329412956255, 2416.
↪0125196180225, -163.36016245424702, -1309.7330667771344, -2170.4745203525963, -1094.
↪9726480270429, 2123.26458714393, 2416.0126942531097, -163.3602301351151, -1309.
↪733210696227, -2170.474634870388, -1094.9726875020344, 2123.2647662400955, 2456.
↪6075093447926, -77.22909111556126, -1283.252068702874, -2130.2151765650274, -1159.
↪8081395939932, 2145.242676822373, -1161.1768654509938, 2145.138477965435, -1280.
↪4300435279195, -2113.915974006277, -1160.6073109958113, 2140.403981134188, -1281.
↪0373135073385, -2109.180381738326, 2415.769857096313, -163.5069028811886, -1309.
↪832637909631, -2170.4632089820057, -1094.7636605550108, 2123.186733078666, 2401.
↪861193733008, -172.353118727211, -1318.579879780884, -2171.2963607061065, -1093.
↪1900162376958, 2120.0426886068794, 2411.6393197737566, -163.81660155967867, -1089.
↪5319085584983, 2121.576761353844, -1082.6282761181412, 2120.6709717842764, 2399.
↪6753950529396, -160.25426417938482, 278.27148663424725, -10.493493912035552, -146.
↪96678765905787, -241.22429003185408, -130.42325689990824, 244.48803229938719, 206.
↪94129996491046, -35.38505415088836, -142.73287289392556, -183.59574244679482, -91.
↪6887699449892, 207.03653181237675, 2401.7765548121, -0.0011773940893547467, -1200.
↪8892060353498, -2079.9993861623666, -1200.8871497819646, 2080.0002010072703, 278.
↪77068751378476, -18.84925622382503, -151.12305792697495, -250.4393740486974, -126.
↪34299926170385, 244.99208111611046]
```

CHAPTER 4

src

4.1 py_dss_interface package

4.1.1 ActiveClass

Created by eniocc at 11/10/2020

class py_dss_interface.models.ActiveClass.ActiveClass.**ActiveClass**(*obj_dss*)
Bases: py_dss_interface.models.ActiveClass.ActiveClassS.ActiveClassS, py_dss_interface.models.ActiveClass.ActiveClassI.ActiveClassI, py_dss_interface.models.ActiveClass.ActiveClassV.ActiveClassV

This interface implements the ActiveClass (IActiveClass) interface of OpenDSS by declaring 3 procedures for accessing the different properties included in this interface: ActiveClassS, ActiveClassI, ActiveClassV. Ih the original paper Davis cited that are 4 procedures, but only 3 were described.

class py_dss_interface.models.ActiveClass.ActiveClass.**ActiveClassI**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/modify the properties of the ActiveClass Class where the values are integers.

The structure of the interface is as follows: int32_t ActiveClassI(int32_t Parameter,int32_t argument).

This interface returns an integer (signed 32 bits), the variable "parameter" is used to specify the property of the class to be used and the variable "argument" can be used to modify the value of the property when necessary. Reading and writing properties are separated and require a different parameter number to be executed.

The properties (parameter) are integer numbers and are described as follows.

active_class_count() → int
Gets the number of elements in this class. Same as NumElements Property.

active_class_first() → int
Sets first element in the active class to be the active DSS object. If object is a CktElement, ActiveCktElement also points to this element. Returns 0 if none.

active_class_next () → int
Sets next element in the active class to be the active DSS object. If object is a CktElement, ActiveCktElement also points to this element. Returns 0 if none.

active_class_num_elements () → int
Gets the number of elements in this class. Same as Count Property.

class py_dss_interface.models.ActiveClass.ActiveClass.**ActiveClassS**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/modify the properties of the ActiveClass Class where the values are strings.

The structure of the interface is as follows: CStr ActiveClassI(int32_t Parameter, CStr argument)

This interface returns a string, the first parameter is used to specify the property of the class to be used and the second parameter can be used to modify the value of the property when necessary. Reading and writing properties are separated and require a different parameter number to be executed.

The properties (parameter) are integer numbers and are described as follows.

active_class_get_class_name () → str
Sets the name of the active Element of the Active class.

active_class_get_name () → str
Gets the name of the active Element of the Active class.

active_class_parent_class_name () → str
Gets the name of the Parent Element of the Active class.

active_class_write_name (*argument*) → str
Sets the name of the active Element of the Active class.

class py_dss_interface.models.ActiveClass.ActiveClass.**ActiveClassV**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/modify the properties of the ActiveClass Class where the values are Variants.

The structure of the interface is as follows: void ActiveClassI(int32_t Parameter, VARIANT *Argument)

This interface returns a Variant, the first parameter is used to specify the property of the class to be used and the second parameter can be used to modify the value of the property when necessary. Reading and writing properties are separated and require a different parameter number to be executed.

The properties (parameter) are integer numbers and are described as follows.

active_class_all_names () → List[str]
Gets a variant array of strings consisting of all element names in the active Class.

4.1.2 Bus

Created by eniocc at 11/10/2020

class py_dss_interface.models.Bus.Bus.**Bus**(*obj_dss*)
Bases: py_dss_interface.models.Bus.BusS.BusS, py_dss_interface.models.Bus.BusI.BusI, py_dss_interface.models.Bus.BusV.BusV, py_dss_interface.models.Bus.BusF.BusF

This interface implements the Bus (IBus) interface of OpenDSS by declaring 4 procedures for accessing the different properties included in this interface: BusS, BusI, BusV, BusF.

class py_dss_interface.models.Bus.Bus.**BusF**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: double BUSF(int32_t Parameter, double Argument)

This interface returns a floating point number (64 bits) according to the number sent in the variable "parameter". The parameter can be one of the following.

bus_distance() → float
Returns the distance from the energymeter (if non-zero).

bus_interruptions_avg_duration() → float
Returns the average interruption duration in hours.

bus_interruptions_num() → float
Returns the number of interruptions this bus per year.

bus_interruptions_total_customers() → float
Returns the annual number of customer interruptions from this bus.

bus_kv_base() → float
Returns the base voltage at bus in kV.

bus_lambda() → float
Returns the accumulated failure rate downstream from this bus, faults per year.

bus_line_total_miles() → float
Returns the total length of line downline from this bus, in miles. For recloser siting algorithm.

bus_outage_customer_accum_duration() → float
Returns the accumulated customer outage durations.

bus_read_latitude() → float
This parameter returns the GIS latitude assigned to the active bus (if any).

bus_read_longitude() → float
This parameter returns the GIS longitude assigned to the active bus (if any).

bus_read_x() → float
Returns the X coordinate for the bus.

bus_read_y() → float
Returns the X coordinate for the bus.

bus_write_latitude(*latitude_param: float*) → float
This parameter sets the GIS latitude to the active bus using the value given at the argument..

bus_write_longitude(*longitude_param: float*) → float
This parameter sets the GIS longitude to the active bus using the value given at the argument..

bus_write_x(*param_coordinate*) → int
Allows to write the X coordinate for the bus. Returns 0. :param param_coordinate: The X coordinate, if it's None, X will be 0.0

bus_write_y(*param_coordinate: float*) → int
Allows to write the Y coordinate for the bus. Returns 0. :param param_coordinate: The Y coordinate, if it's None, Y will be 0.0

class py_dss_interface.models.Bus.Bus.**BusI**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: int32_t BUSI(int32_t Parameter, int32_t Argument)

This interface returns an integer according to the number sent in the variable "parameter". The parameter can be one of the following.

bus_coord_defined() → int
Returns 1 if a coordinate has been defined for this bus; otherwise, it will return 0.

bus_get_unique_node_number(*start_number: int = 1*) → int
Returns a unique node number at the active bus to avoid node collisions and adds it to the node list for the bus. The start number can be specified in the second parameter.

Parameters **start_number** – The first number corresponding the initial bus node number

Returns int

Return type int

bus_num_nodes() → int
Returns the number of nodes of this bus.

bus_section_id() → int
Returns the integer ID of the feeder section in which this bus is located.

bus_total_customers() → int
Returns returns the total number of customers served down line from this bus.

bus_zsc_refresh() → int
Recomputes Zsc for active bus for present circuit configuration. Return 1 if the procedure was successful.

class py_dss_interface.models.Bus.Bus.**BusS**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: CStr BUSS(int32_t Parameter, CStr Argument)

This interface returns a string according to the number sent in the variable "parameter". The parameter can be one of the following.

bus_name() → str
Returns the name of the active bus.

class py_dss_interface.models.Bus.Bus.**BusV**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: void BUSV(int32_t Parameter, VARIANT *Argument)

This interface returns a variant according to the number sent in the variable "parameter". The parameter can be one of the following.

bus_all_pce_active_bus() → List[str]
Returns an array with the names of all PCE connected to the active bus.

bus_all_pde_active_bus() → List[str]
Returns an array with the names of all PDE connected to the active bus.

bus_axc_012_matrix()
Variant array of doubles (complex) containing the complete 012 Zsc matrix.

bus_cplx_sequence_voltages()
Returns the complex double array of sequence voltages (0, 1, 2) at this bus.

bus_isc()
Returns the short circuit current as complex array.

bus_line_list()
This parameter returns a variant array of strings containing the names of the lines connected to the active bus. The names of the lines include the class name 'Line.'

bus_load_list()
This parameter returns a variant array of strings containing the names of the loads connected to the active bus. The names of the lines include the class name 'Load.'.

bus_nodes() → List[int]
Returns an integer array of node numbers defined at the bus in same order as the voltages.

bus_pu_vll()
Returns a variant array of complex numbers representing L-L voltages in per unit. Returns -1.0 for 1-phase bus. If more than 3 phases, returns only first 3..

bus_pu_vmag_angle()
Returns a variant array of doubles containing voltages in per unit and angles in degrees.

bus_pu_voltages()
Returns the voltages in per unit at bus as complex array.

bus_seq_voltages()
Returns a complex array of Sequence voltages at this bus.

bus_vll()
For 2 and 3 phase buses, returns a variant array of complex numbers representing L-L voltages in volts. Returns -1.0 for 1-phase bus. If more than 3 phases, returns only first 3.

bus_vmag_angle()
Returns a variant array of doubles containing voltages in magnitude (VLN), angle (deg).

bus_voc()
Returns the open circuit voltage as complex array.

bus_voltages()
Returns a complex array of voltages at this bus.

bus_ysc_matrix()
Returns the complex array of Ysc matrix at bus, column by column.

bus_zsc0()
Returns the complex zero-sequence short circuit impedance at bus.

bus_zsc1()
Returns the complex array of Zsc matrix at bus, column by column.

bus_zsc_matrix()
Returns the complex array of Zsc matrix at bus, column by column.

4.1.3 Capacitors

Created by eniocc at 11/10/2020

class py_dss_interface.models.Capacitors.Capacitors.**Capacitors**(*obj_dss*)
Bases: py_dss_interface.models.Capacitors.CapacitorsF.CapacitorsF, py_dss_interface.models.Capacitors.CapacitorsI.CapacitorsI, py_dss_interface.models.Capacitors.CapacitorsS.CapacitorsS, py_dss_interface.models.Capacitors.CapacitorsV.CapacitorsV

This interface implements the Capacitors (ICapacitors) interface of OpenDSS by declaring 4 procedures for accessing the different properties included in this interface: CapacitorsF, CapacitorsI, CapacitorsS, CapacitorsV.

class py_dss_interface.models.Capacitors.Capacitors.**CapacitorsF**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/modify the properties of the Capacitors Class where the values are doubles.

The structure of the interface is as follows: double CapacitorsF(int32_t Parameter, double argument)

This interface returns a floating point number (64 bits), the first parameter is used to specify the property of the class to be used and second parameter can be used to modify the value of the property when necessary. Reading and writing properties are separated and require a different parameter number to be executed.

The properties (parameter) are integer numbers and are described as follows.

capacitors_read_kv() → float
Gets the bank rating. Use LL for 2 or 3 phases, or actual can rating for 1 phase.

capacitors_read_kvar() → float
Gets the total bank kvar, distributed equally among phases and steps.

capacitors_write_kv(*argument: float*)
Sets the bank rating. Use LL for 2 or 3 phases, or actual can rating for 1 phase. There is not a explicit return type in the oficial documentation, because of this we choose not put a explicit return too.

capacitors_write_kvar(*argument: float*)
Sets the total bank kvar, distributed equally among phases and steps. There is not a explicit return type in the oficial documentation, because of this we choose not put a explicit return too.

class py_dss_interface.models.Capacitors.Capacitors.**CapacitorsI**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/modify the properties of the Capacitors Class where the values are integers.

The structure of the interface is as follows: int32_t CapacitorsI(int32_t Parameter, int32_t argument)

This interface returns an integer (signed 32 bits), the variable "parameter" is used to specify the property of the class to be used and the variable "argument" can be used to modify the value of the property when necessary. Reading and writing properties are separated and require a different parameter number to be executed.

The properties (parameter) are integer numbers and are described as follows.

capacitors_add_step() → int
Adds one step of the capacitor if available. If successful returns 1.

capacitors_available_steps() → int
Gets the number of steps available in cap bank to be switched ON.

capacitors_close() → int
Closes all steps, all phases of the capacitor.

capacitors_count() → int
Gets the number of capacitor objects in active circuit.

capacitors_first() → int
Sets the first capacitor active. Returns 0 if no more.

capacitors_next() → int
Sets the next capacitor active. Returns 0 if no more.

capacitors_open() → int
Opens all steps, all phases of the capacitor.

capacitors_read_is_delta() → int
Gets 1 if delta connection, otherwise will return 0 for distributing and switching the total kvar.

capacitors_read_num_steps() → int
Gets the number of steps (defaults 1) for distributing and switching the total bank kvar.

capacitors_subtract_step() → int
Subtracts one step of the capacitor if available. If no more steps, returns 0.

capacitors_write_is_delta(*argument: int = 1*) → int
Sets (Argument) 1 if delta connection, otherwise will return 0 for distributing and switching the total kvar.

capacitors_write_num_steps(*argument: int*) → int
Sets the number of steps (defaults 1) for distributing and switching the total bank kvar.

class py_dss_interface.models.Capacitors.Capacitors.**CapacitorsS**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/modify the properties of the Capacitors Class where the values are Strings.

The structure of the interface is as follows: CStr CapacitorsS(int32_t Parameter, CStr capacitor_name)

This interface returns a string, the first parameter is used to specify the property of the class to be used and the second parameter can be used to modify the value of the property when necessary. Reading and writing properties are separated and require a different parameter number to be executed.

The properties (parameter) are integer numbers and are described as follows.

capacitors_read_name() → str
Gets the name of the active Capacitor element.

capacitors_write_name(*capacitor_name: str*) → str
Sets the name of the Capacitor element to set it active. There is not a explicit return type in the oficial documentation, because of this we choose not put a explicit return too. :param capacitor_name: the intended name to the capacitor

class py_dss_interface.models.Capacitors.Capacitors.**CapacitorsV**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/modify the properties of the Capacitors Class where the values are Variants.

The structure of the interface is as follows: void CapacitorsV(int32_t Parameter, VARIANT *Argument)

This interface returns a Variant, the first parameter is used to specify the property of the class to be used and the second parameter can be used to modify the value of the property when necessary. Reading and writing properties are separated and require a different parameter number to be executed.

The properties (parameter) are integer numbers and are described as follows.

capacitors_all_names() → List[str]
Gets a variant array of strings with all Capacitor names in the circuit.

capacitors_read_states() → List[int]
Gets a variant array of integers [0..numsteps-1] indicating the state of each step. If value is -1 and error has occurred.

capacitors_write_states(*dss_*, *argument: list*) → int
Sets a variant array of integers [0..numsteps-1] indicating the state of each step. If value is -1 and error has occurred. :param **dss_**: an instance of dss object :param argument: list with status of Capacitor states

4.1.4 CapControls

Created by eniocc at 11/10/2020

class py_dss_interface.models.CapControls.CapControls.**CapControls**(*obj_dss*)
Bases: py_dss_interface.models.CapControls.CapControlsF.CapControlsF, py_dss_interface.models.CapControls.CapControlsI.CapControlsI, py_dss_interface.models.CapControls.CapControlsS.CapControlsS, py_dss_interface.models.CapControls.CapControlsV.CapControlsV

This interface implements the CapControls (ICapControls) interface of OpenDSS by declaring 4 procedures for accessing the different properties included in this interface: CapControlsF, CapControlsI, CapControlsS, CapControlsV

class py_dss_interface.models.CapControls.CapControls.**CapControlsF**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: double CapControlsF(int32_t Parameter, double Argument);

This interface returns a floating point number (64 bits) with the result of the query according to the value of the variable Parameter, which can be one of the following.

capcontrols_read_ct_ratio() → float
Gets the transducer ratio current to control current.

capcontrols_read_dead_time() → float
Gets the time delay [s] after switching off a step. Control may reset before actually switching.

capcontrols_read_delay() → float
Gets the time delay [s] to switch on after arming. Control may reset before actually switching.

capcontrols_read_delay_off() → float
Gets the time delay [s] before switching off a step. Control may reset before actually switching.

capcontrols_read_off_setting() → float
Gets the threshold to switch off a step. See Mode for Units.

capcontrols_read_on_setting() → float
Gets the threshold to arm or switch on a step. See Mode for Units.

capcontrols_read_pt_ratio() → float
Gets the transducer ratio from primary feeder to control voltage.

capcontrols_read_vmax() → float
Gets the Vmax, this reference with VoltOverride, switch off whenever PT voltage exceeds this level.

capcontrols_read_vmin() → float
Gets the Vmin, this reference with VoltOverride, switch ON whenever PT voltage drops below this level.

capcontrols_write_ct_ratio(*argument: float*) → float
Sets the transducer ratio current to control current.

capcontrols_write_dead_time(*argument: float*) → float
Sets the time delay [s] after switching off a step. Control may reset before actually switching..

capcontrols_write_delay(*argument: float*) → float
Sets the time delay [s] to switch on after arming. Control may reset before actually switching.

capcontrols_write_delay_off(*argument: float*) → float
Sets the time delay [s] before switching off a step. Control may reset before actually switching.

capcontrols_write_off_setting(*argument: float*) → float
Sets the threshold to switch off a step. See Mode for Units.

capcontrols_write_on_setting(*argument: float*) → float
Sets the threshold to arm or switch on a step. See Mode for Units.

capcontrols_write_pt_ratio(*argument: float*) → float
Sets the transducer ratio from primary feeder to control voltage.

capcontrols_write_vmax(*argument: float*) → float
Sets the Vmax, this reference with VoltOverride, switch off whenever PT voltage exceeds this level.

capcontrols_write_vmin(*argument: float*) → float
Sets the Vmin, this reference with VoltOverride, switch ON whenever PT voltage drops below this level.

class py_dss_interface.models.CapControls.CapControls.**CapControlsI**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: int32_t CapControlsI(int32_t Parameter, int32_t Argument);

This interface returns an integer with the result of the query according to the value of the variable Parameter, which can be one of the following

capcontrols_count() → int
Gets the number of CapControls in Active Circuit.

capcontrols_first() → int
Sets the first CapControl active. Returns 0 if no more.

capcontrols_next() → int
Sets the next CapControl active. Returns 0 if no more.

capcontrols_read_mode() → int
Gets the type of automatic controller (see manual for details). CURRENTCONTROL: Result := 0; VOLTAGECONTROL: Result := 1; VARCONTROL: Result := 2; TIMECONTROL: Result := 3; PFCONTROL: Result := 4; USERCONTROL: Result := 4;

capcontrols_read_monitored_term() → int
Gets the terminal number on the element that PT and CT are connected to.

capcontrols_read_use_volt_override() → int
Gets if Vmin and Vmax are enabled to override the control Mode. There is not a explicit return type in the oficial documentation, because of this we choose not put a explicit return too." return self.dss_obj.CapControlsI(6, 0).

capcontrols_write_mode(*argument: int*) → int
Sets the type of automatic controller (see manual for details). 0: elem.CapControlType := CURRENTCONTROL; 1: elem.CapControlType := VOLTAGECONTROL; 2: elem.CapControlType := KVARCONTROL; 3: elem.CapControlType := TIMECONTROL; 4: elem.CapControlType := PFCONTROL;

capcontrols_write_monitored_term(*dss*, *argument: int*) → int
Sets the terminal number on the element that PT and CT are connected to. There is not a explicit return type in the oficial documentation, because of this we choose not put a explicit return too.

capcontrols_write_use_volt_override(*dss*, *argument: int*) → int
Sets if enables Vmin and Vmax to override the control Mode. There is not a explicit return type in the oficial documentation, because of this we choose not put a explicit return too.

class py_dss_interface.models.CapControls.CapControls.**CapControlsS**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: CStr CapControlsS(int32_t Parameter, CStr Argument);

This interface returns a string with the result of the query according to the value of the variable Parameter, which can be one of the following.

capcontrols_read_capacitor() → str
Gets the name of the capacitor that is controlled.

capcontrols_read_monitored_obj() → str
Gets the full name of the element that PT and CT are connected to.

capcontrols_read_name() → str
Gets the name of the active CapControl.

capcontrols_write_capacitor(*argument: str*) → str
Sets the name of the capacitor that is controlled.

capcontrols_write_monitored_obj(*argument: str*) → str
Sets the full name of the element that PT and CT are connected to.

capcontrols_write_name(*argument: str*) → str
Sets a CapControl active by name.

class py_dss_interface.models.CapControls.CapControls.**CapControlsV**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: void CapControlsV(int32_t Parameter, VARIANT *Argument);

This interface returns a Variant with the result of the query according to the value of the variable Parameter, which can be one of the following.

capcontrols_all_names() → List[str]

4.1.5 Circuit

Created by eniocc at 11/10/2020

class py_dss_interface.models.Circuit.Circuit.**Circuit**(*obj_dss*)
Bases: py_dss_interface.models.Circuit.CircuitI.CircuitI, py_dss_interface.models.Circuit.CircuitS.CircuitS, py_dss_interface.models.Circuit.CircuitF.CircuitF, py_dss_interface.models.Circuit.CircuitV.CircuitV

This interface implements the Circuit (ICIrcuit) interface of OpenDSS by declaring 4 procedures for accessing the different properties included in this interface: CircuitI, CircuitS, CircuitF, CircuitV

class py_dss_interface.models.Circuit.Circuit.**CircuitF**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: double CircuitF(int32_t Parameter, double Argument1, double Argument2);

This interface returns a floating point number (IEEE754 64 bits) according to the number sent in the variable "parameter". The parameter can be one of the following.

circuit_capacity(*capacity_start=0*, *capacity_increment=0.1*) → float
Returns the total capacity of the active circuit. Or this parameter it is necessary to specify the start and increment of the capacity in the arguments argument1 and argument2 respectively.

circuit_float(*first*, *second*, *third*) → float

class py_dss_interface.models.Circuit.Circuit.**CircuitI**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: int32_t CircuitI(int32_t Parameter, int32_t Argument);

This interface returns an integer according to the number sent in the variable "parameter". The parameter can be one of the following.

circuit_end_of_time_step_update() → int
Calls end of time step cleanup routine in solutionalgs.pas. Returns 0.

circuit_first_element() → int
Sets the first Element of the active class to be the active Element, as a result, this parameter will deliver the index of the active Element (0 if none).

circuit_first_pc_element() → int
Sets the first PCElement to be the active PCElement, as a result, this parameter will deliver the index of the active PCElement (ideally 1).

circuit_first_pd_element() → int
Sets the first PDElement to be the active PDElement, as a result, this parameter will deliver the index of the active PDElement (ideally 1).

circuit_integer(*first: int*, *second: int*) → int

circuit_next_element() → int
Sets the next Element of the active class to be the active Element, as a result, this parameter will deliver the index of the active Element (0 if none).

circuit_next_pc_element() → int
Sets the next PCElement to be the active PCElement, as a result, this parameter will deliver the index of the active PCElement (if there is no more it will return a 0).

circuit_next_pd_element() → int
Sets the next PDElement to be the active PDElement, as a result, this parameter will deliver the index of the active PDElement (if there is no more it will return a 0).

circuit_num_buses() → int
Will deliver the number of buses included in the active circuit.

circuit_num_ckt_elements() → int
Will deliver the number of CktElements included in the active circuit.

circuit_num_nodes() → int
Will deliver the number of nodes included in the active circuit.

circuit_parent_pd_element() → int
Sets parent PD Element, if any, to be the active circuit element and returns index > 0 if it fails or not applicable.

circuit_sample() → int
Forces all meters and monitors to take a sample, returns 0.

circuit_save_sample() → int
Forces all meters and monitors to save their sample buffers, returns 0.

circuit_set_active_bus_i(*i: int*) → int
Sets active the bus specified by index, which is compatible with the index delivered by AllBusNames, returns 0 it everything ok.

circuit_update_storage_t()
Forces all storage classes to update. Typically done after a solution.

class py_dss_interface.models.Circuit.Circuit.**CircuitS**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: CStr CircuitS(int32_t Parameter, CStr Argument);

This interface returns a string according to the number sent in the variable "parameter". The parameter can be one of the following.

circuit_disable() → str
Allows to disable an element of the active circuit, the element must be specified by name. As a result, this parameter will deliver the string ?Ok?.

circuit_enable() → str
Allows to enable an element of the active circuit, the element must be specified by name. As a result, this parameter will deliver the string ?Ok?.

circuit_name() → str
Returns the name of the active circuit.

circuit_set_active_bus(*argument: str*) → str
Allows to activate a bus of the active circuit, the bus must be specified by name. As a result, this parameter will deliver a string with the index of the active Bus.

circuit_set_active_class(*argument: str*) → str
Allows tto activate a Class of the active circuit, the Class must be specified by name. As a result, this parameter will deliver a string with the index of the active Class.

circuit_set_active_element(*argument: str*) → str
Allows to activate an element of the active circuit, the element must be specified by name. As a result, this parameter will deliver a string with the index of the active element.

class py_dss_interface.models.Circuit.Circuit.**CircuitV**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: void CircuitV(int32_t Parameter, VARIANT *Argument, int32_t Argument2);

This interface returns a variant according to the number sent in the variable "parameter". The parameter can be one of the following.

circuit_all_bus_distances() → List[float]
Returns distance from each bus to parent EnergyMeter. Corresponds to sequence in AllBusNames. Argument2 must be 0.

circuit_all_bus_names() → List[str]
Returns an array of strings with the names of all the Buses of the active circuit (See AllNodeNames). Argument2 must be 0.

circuit_all_bus_vmag() → List[float]
Returns an array of doubles (magnitude) with the node voltages from the most recent solution. Argument2 must be 0.

circuit_all_bus_vmag_pu() → List[float]
Returns an array of doubles with the voltages in per unit of the most recent solution of the active circuit. Argument2 must be 0.

circuit_all_bus_volts() → List[float]
Returns an array of doubles (two doubles for representing a complex number) with the node voltages from the most recent solution. Argument2 must be 0.

circuit_all_element_losses() → List[float]
Returns an array of doubles (two doubles for representing a complex number) with the losses in each element of the active circuit. Argument2 must be 0.

circuit_all_element_names() → List[str]
Returns an array of strings with the names of all the elements of the active circuit. Argument2 must be 0.

circuit_all_node_distances() → List[float]
Returns distance from each Node to parent EnergyMeter. Corresponds to sequence in AllBusVmag. Argument2 must be 0.

circuit_all_node_distances_by_phase(*argument: int = 1*) → List[float]
Returns array of doubles representing the distances to parent EnergyMeter. Sequence of array corresponds to other node ByPhase properties. Argument2 must contain the number of the phase to return.

circuit_all_node_names() → List[str]
Returns an array of strings containing full name of each node in system in same order as returned by AllBusVolts, etc. Argument2 must be 0.

circuit_all_node_names_by_phase(*argument: int = 1*) → List[str]
Returns array of strings of the node names by Phase criteria. Sequence corresponds to other ByPhase properties. Argument2 must contain the number of the phase to return.

circuit_all_node_vmag_by_phase(*argument: int = 1*) → List[float]
Returns array of doubles representing the voltage magnitudes for nodes on the specified phase. The phase must be specified in the Argument2.

circuit_all_node_vmag_pu_by_phase(*argument: int = 1*) → List[float]
Returns array of doubles representing the voltage magnitudes (in per unit) for nodes on the specified phase. The phase must be specified in the Argument2.

circuit_line_losses() → List[float]
Returns an array of doubles (two doubles for representing a complex number) with the total Line losses of the active circuit. Argument2 must be 0.

circuit_losses() → List[float]
Returns an array of doubles (two doubles for representing a complex number) with the total losses of the active circuit. Argument2 must be 0.

circuit_substation_losses() → List[float]
Returns an array of doubles (two doubles for representing a complex number) with the total transformer losses of the active circuit. Argument2 must be 0.

circuit_system_y() → List[float]
Returns an array of doubles (two doubles for representing a complex number) containing the Y Bus Matrix of the system (after a solution has been performed). Argument2 must be 0.

circuit_total_power() → List[float]
Returns an array of doubles (two doubles for representing a complex number) with the total power in watts delivered to the active circuit. Argument2 must be 0.

circuit_y_currents()
Returns a variant array of doubles containing complex injection currents for the present solution. It is the "I" vector of I=YV. Argument2 must be 0.

circuit_y_node_order() → List[str]
Returns a variant array of strings containing the names of the nodes in the same order as the Y Matrix. Argument2 must be 0.

circuit_y_node_varray()
Returns a complex array of actual node voltages in same order as SystemY Matrix. Argument2 must be 0.

4.1.6 CktElement

Created by eniocc at 11/10/2020

class py_dss_interface.models.CktElement.CktElement.**CktElement**(*obj_dss*)
Bases: py_dss_interface.models.CktElement.CktElementI.CktElementI, py_dss_interface.models.CktElement.CktElementS.CktElementS, py_dss_interface.models.CktElement.CktElementF.CktElementF, py_dss_interface.models.CktElement.CktElementV.CktElementV

This interface implements the CktElement interface of OpenDSS by declaring 4 procedures for accessing the different properties included in this interface: CktElementI, CktElementS, CktElementF, CktElementV

class py_dss_interface.models.CktElement.CktElement.**CktElementF**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

cktelement_read_emerg_amps() → float
Deliver the Emergency ampere rating for the active PDElement.

cktelement_read_norm_amps() → float
Deliver the normal ampere rating for the active PDElement.

cktelement_variable_i(*argument: float*) → float
Delivers get the value of a variable by index for the active PCElement.

cktelement_write_emerg_amps(*argument: float*) → float
Allows to fix the Emergency ampere rating for the active PDElement. The new value must be defined in the variable ?Argument?.

cktelement_write_norm_amps(*argument: float*) → float
Allows to fix the normal ampere rating for the active PDElement.

class py_dss_interface.models.CktElement.CktElement.**CktElementI**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: int32_t CktElementI(int32_t Parameter, int32_t Argument);

This interface returns an integer with the result of the query according to the value of the variable Parameter, which can be one of the following.

cktelement_close(*argument: int*) → int
Close the specified terminal (Argument) of the active DSS object.

cktelement_has_switch_control() → int
Returns 1 if the active DSS object has a Switch Control linked; otherwise, it will return 0.

cktelement_has_volt_control() → int
Returns 1 if the active DSS object has a Volt Control linked; otherwise, it will return 0.

cktelement_is_open() → int
Return a 1 if any terminal of the active DSS object is open, otherwise, it will return a 0.

cktelement_num_conductors() → int
Deliver the number of conductors of the active DSS object.

cktelement_num_controls() → int
Returns number of controls linked to the active DSS object.

cktelement_num_phases() → int
Delivers the number of phases of the active DSS object.

cktelement_num_properties() → int
Return the number of properties of the active DSS object.

cktelement_num_terminals() → int
Deliver the number of terminals of the active DSS object.

cktelement_ocp_dev_index() → int
Returns the Index into Controller list of OCP Device controlling the active DSS object.

cktelement_ocp_dev_type() → int
Returns one of the following values: 0=none; 1=Fuse; 2=Recloser; 3=Relay according to the type of active control.

cktelement_open(*argument: int*) → int
Open the specified terminal (Argument) of the active DSS object.

cktelement_read_enabled() → int
Returns one of the following values: 0 if the active element is disabled or 1 if the active element is enabled.

cktelement_write_enabled(*argument: int*) → int
Returns one of the following values: 0 if the active element is disabled or 1 if the active element is enabled.

class py_dss_interface.models.CktElement.CktElement.**CktElementS**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: CStr CktElementS(int32_t Parameter, CStr Argument);

This interface returns a string (pAnsiChar) with the result of the query according to the value of the variable Parameter, which can be one of the following.

cktelement_controller(*argument: str*) → str
Delivers the Full name of the i-th controller attached to the active circuit element. The i-th controller index must be specified in the argument arg. Ex: Str = Controller(2). See NumControls to determine valid index range.

cktelement_energymeter() → str
Delivers the name of the EnergyMeter linked to the active circuit element.

cktelement_guid() → str
Delivers the unique name for the active circuit element.

cktelement_name() → str
Delivers the full name of the active circuit element.

cktelement_read_display() → str
Displays the name of the active circuit element (not necessarily unique).

cktelement_write_display(*argument: str*) → str
Allows to modify the name of the active circuit element (not necessarily unique).

class py_dss_interface.models.CktElement.CktElement.**CktElementV**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: void CktElementV(int32_t Parameter, VARIANT *Argument);

This interface returns a variant (the format depends on the parameter) with the result of the query according to the value of the variable Parameter, which can be one of the following.

cktelement_all_property_names() → List[str]
Delivers an array of strings with the names of all the properties of the active circuit element.

cktelement_all_variables_names() → List[str]
Delivers a Variant array of strings listing all the published state variable names, if the active circuit element is a PCElement. Otherwise, null string.

cktelement_all_variables_values() → List[float]
Delivers a Variant array of doubles listing all the values of the state variables, if the active circuit element is a PCElement. Otherwise, null string.

cktelement_cplx_seq_currents() → List[float]
Delivers an array of doubles with the complex of sequence currents for all terminals of the active circuit element.

cktelement_cplx_seq_voltages() → List[float]
Delivers an array of doubles with the complex of sequence voltages for all terminals of the active circuit element.

cktelement_currents() → List[float]
Delivers an array of doubles with the currents at terminals of the active circuit element.

cktelement_currents_mag_ang() → List[float]
Delivers the currents in magnitude, angle format as a variant array of doubles of the active circuit element.

cktelement_losses() → List[float]
Delivers an array of doubles with the Losses at terminals of the active circuit element.

cktelement_node_order() → List[int]
Delivers a Variant array integers variant array of integer containing the node numbers (representing phases, for Example) for each conductor of each terminal.

cktelement_phase_losses() → List[float]
Delivers an array of doubles with the Losses per phase at the terminals of the active circuit element.

cktelement_powers() → List[float]
Delivers an array of doubles with the powers at terminals of the active circuit element.

cktelement_read_bus_names() → List[str]
Delivers an array of strings with the names of all the buses connected to the active circuit element.

cktelement_residuals() → List[float]
Delivers an array of doubles with the residual currents (magnitude, angle) in all the nodes of the active circuit element.

cktelement_seq_currents() → List[float]
Delivers an array of doubles with the symmetrical component Currents per phase at the terminals of the active circuit element.

cktelement_seq_powers() → List[float]
Delivers an array of doubles with the symmetrical component powers per phase at the terminals of the active circuit element.

cktelement_seq_voltages() → List[float]
Delivers an array of doubles with the symmetrical component voltages per phase at the terminals of the active circuit element.

cktelement_voltages() → List[float]
Delivers an array of doubles with the voltages at terminals of the active circuit element.

cktelement_voltages_mag_ang() → List[float]
Delivers the voltages in magnitude, angle format as a variant array of doubles of the active circuit element.

cktelement_write_bus_names(*dss, argument: List[str]*) → str
Allows to fix an array of strings with the names of all the buses connected to the active circuit element.

cktelement_y_prim() → List[float]
Delivers an array of doubles with the Y primitive matrix (complex) of the active circuit element.

4.1.7 CMathLib

Created by eniocc at 11/10/2020

class py_dss_interface.models.CMathLib.CMathLib.**CMathLib**(*obj_dss*)
Bases: py_dss_interface.models.CMathLib.CMathLibF.CMathLibF, py_dss_interface.models.CMathLib.CMathLibV.CMathLibV

This interface implements the CmathLib (ICmathLib) interface of OpenDSS by declaring 2 procedures for accessing the different properties included in this interface: CMathLibF, CMathLibV.

class py_dss_interface.models.CMathLib.CMathLib.**CMathLibF**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: double CmathLibF(int32_t Parameter, double Argument1, double Argument2);

This interface returns a floating point number with the result of the query according to the value of the variable Parameter, which can be one of the following.

cmathlib_cabs(*arg_real: float*, *arg_imaginary: float*) → float
Returns the absolute value of complex number given in real (Argument1) and imaginary (Argument2) doubles.

cmathlib_cdang(*arg_real: float*, *arg_imaginary: float*) → float
Returns the angle, in degrees, of a complex number specified as two doubles: Real part (Argument1) and imaginary part (Argument2).

class py_dss_interface.models.CMathLib.CMathLib.**CMathLibV**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: void CmathLibV(int32_t Parameter, double Argument1, Argument2, VARIANT *Argument);

This interface returns a Variant with the result of the query according to the value of the variable Parameter, which can be one of the following.

cmathlib_cmplx(*real_part: float*, *imag_part: float*) → complex
Convert real (Argument1) and imaginary (Argument1) doubles to variant array of doubles.

cmathlib_ctopolardeg(*double_real: float*, *double_imag: float*) → Tuple[float, float]
Convert complex number (Argument1 and Argument2) to magnitude and angle, degrees. Returns variant array of two doubles.

cmathlib_pdegtocomplex(*double_real: float*, *double_imag: float*) → complex
Convert magnitude, angle in degrees (Argument1 and Argument2) to a complex number. Returns variant array of two doubles.

4.1.8 CtrlQueue

Created by eniocc at 11/10/2020

class py_dss_interface.models.CtrlQueue.CtrlQueue.**CtrlQueue**(*obj_dss*)
Bases: py_dss_interface.models.CtrlQueue.CtrlQueueI.CtrlQueueI, py_dss_interface.models.CtrlQueue.CtrlQueueV.CtrlQueueV

This interface implements the CtrlQueue (ICtrlQueue) interface of OpenDSS by declaring 2 procedures for accessing the different properties included in this interface: CtrlQueueI, CtrlQueueV

class py_dss_interface.models.CtrlQueue.CtrlQueue.**CtrlQueueI**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/modify the properties of the CtrlQueue Class where the values are integers.

The structure of the interface is as follows: int32_t CtrlQueueI(int32_t Parameter, int32_t argument)

This interface returns an integer (signed 32 bits), the variable "parameter" is used to specify the property of the class to be used and the variable "argument" can be used to modify the value of the property when necessary. Reading and writing properties are separated and require a different parameter number to be executed.

The properties (parameter) are integer numbers and are described as follows.

ctrlqueue_action(*argument: int*) → int
Sets the active action by index (argument).

ctrlqueue_action_code() → int
Gets the code for the active action. Long integer code to tell the control device what to do.

ctrlqueue_clear_actions() → int
Clears the action list.

ctrlqueue_clear_queue() → int
Clears the control queue.

ctrlqueue_delete(*argument: int*) → int
Deletes a control action from the DSS control queue by referencing the handle of the action (Argument).

ctrlqueue_device_handle() → int
Gets the handle (user defined) to device that must act on the pending action.

ctrlqueue_do_all_queue() → int
Forces the execution of all control actions stored at the control queue. Returns 0.

ctrlqueue_num_actions() → int
Gets the number of actions on the current action list (that have been popped off the control queue by CheckControlActions).

ctrlqueue_pop_action() → int
Pops next action off the action list and makes it the active action. Returns zero if none.

ctrlqueue_push() → int
Pushes a control action onto the DSS control queue by time, action code, and device handle. Returns Control Queue handle.

ctrlqueue_queue_size() → int
Delivers the size of the current control queue. Returns zero if none.

ctrlqueue_show() → int
Shows the entire control queue in CSV format.

class py_dss_interface.models.CtrlQueue.CtrlQueue.**CtrlQueueV**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/modify the properties of the CtrlQueue Class where the values are Variants.

The structure of the interface is as follows: void CtrlQueueV(int32_t Parameter, , VARIANT *Argument);

This interface returns a Variant, the variable "parameter" is used to specify the property of the class to be used and the variable "argument" can be used to modify the value of the property when necessary. Reading and writing properties are separated and require a different parameter number to be executed.

The properties (parameter) are integer numbers and are described as follows.

ctrlqueue_ctrlqueue() → str
Delivers the control actions contained in the CtrlQueue after the latest solve command.

4.1.9 DSSElement

Created by eniocc at 11/10/2020

class py_dss_interface.models.DSSElement.DSSElement.**DSSElement**(*obj_dss*)
Bases: py_dss_interface.models.DSSElement.DSSElementI.DSSElementI, py_dss_interface.models.DSSElement.DSSElementS.DSSElementS, py_dss_interface.models.DSSElement.DSSElementV.DSSElementV

This interface implements the DSSElement (IDSSElement) interface of OpenDSS by declaring 3 procedures for accessing the different properties included in this interface: DSSElementI, DSSElementS, DSSElementV

class py_dss_interface.models.DSSElement.DSSElement.**DSSElementI**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/modify the properties of the DSSElement Class where the values are integers.

The structure of the interface is as follows: int32_t DSSElementI(int32_t Parameter, int32_t argument) ;

This interface returns an integer (signed 32 bits), the variable "parameter" is used to specify the property of the class to be used and the variable "argument" can be used to modify the value of the property when necessary. Reading and writing properties are separated and require a different parameter number to be executed.

The properties (parameter) are integer numbers and are described as follows.

dsselement_num_properties() → int
Gets the number of properties for the active DSS object.

class py_dss_interface.models.DSSElement.DSSElement.**DSSElementS**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/modify the properties of the DSSElement Class where the values are Strings.

The structure of the interface is as follows: CStr DSSElementS(int32_t Parameter, CStr argument) ;

This interface returns a string, the variable "parameter" is used to specify the property of the class to be used and the variable "argument" can be used to modify the value of the property when necessary. Reading and writing properties are separated and require a different parameter number to be executed.

The properties (parameter) are integer numbers and are described as follows.

dsselement_name() → str
Gets the full name of the active DSS object (general element or circuit element).

class py_dss_interface.models.DSSElement.DSSElement.**DSSElementV**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/modify the properties of the DSSElement Class where the values are Variants.

The structure of the interface is as follows: void DSSElementV(int32_t Parameter VARIANT *Argument) ;

This interface returns a Variant, the variable "parameter" is used to specify the property of the class to be used and the variable "argument" can be used to modify the value of the property when necessary. Reading and writing properties are separated and require a different parameter number to be executed.

The properties (parameter) are integer numbers and are described as follows.

dsselement_all_property_names() → List[str]
Gets a variant array of strings containing the names of all properties for the active DSS object.

4.1.10 DSSExecutive

Created by eniocc at 11/10/2020

class py_dss_interface.models.DSSExecutive.DSSExecutive.**DSSExecutive**(*obj_dss*)
Bases: py_dss_interface.models.DSSExecutive.DSSExecutiveS.DSSExecutiveS, py_dss_interface.models.DSSExecutive.DSSExecutiveI.DSSExecutiveI

This interface implements the DSS_Executive (IDSS_Executive) interface of OpenDSS by declaring 2 procedures for accessing the different properties included in this interface: DSSExecutiveS, DSSExecutiveI

class py_dss_interface.models.DSSExecutive.DSSExecutive.**DSSExecutiveI**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: int32_t DSSExecutiveI(int32_t Parameter, int32_t Argument);

This interface returns an integer with the result of the query according to the value of the variable Parameter, which can be one of the following.

executive_num_commands() → int
Gets the number of DSS Executive Commands.

executive_num_options() → int
Gets the number of DSS Executive Options.

class py_dss_interface.models.DSSExecutive.DSSExecutive.**DSSExecutiveS**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: CStr DSSExecutiveS(int32_t Parameter, CStr Argument);

This interface returns a string with the result of the query according to the value of the variable Parameter, which can be one of the following.

executive_command(*arg: str*) → str
Gets i-th command (specified in the argument as string).

executive_command_help(*arg: str*) → str
Gets help string for i-th command (specified in the argument as string).

executive_option(*arg: str*) → str
Gets i-th option (specified in the argument as string).

executive_option_help(*arg: str*) → str
Gets help string for i-th option (specified in the argument as string).

executive_option_value(*arg: str*) → str
Gets present value for i-th option (specified in the argument as string).

4.1.11 DSSInterface

Created by eniocc at 11/10/2020

class py_dss_interface.models.DSSInterface.DSSInterface.**DSSInterface**(*obj_dss*)
Bases: py_dss_interface.models.DSSInterface.DSSInterfaceS.DSSInterfaceS, py_dss_interface.models.DSSInterface.DSSInterfaceI.DSSInterfaceI, py_dss_interface.models.DSSInterface.DSSInterfaceV.DSSInterfaceV

This interface implements the DSS interface (IDSS - DDSS.pas) of OpenDSS by declaring 3 procedures for accessing the different properties included in this interface: DSSInterfaceS, DSSInterfaceI, DSSInterfaceV

class py_dss_interface.models.DSSInterface.DSSInterface.**DSSInterfaceI**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: int32_t DSSI(int32_t Parameter, int32_t Argument);

This interface returns an integer with the result of the query according to the value of the variable Parameter, which can be one of the following.

dss_clear_all()
Clears all circuit definitions.

dss_num_circuits() → int
Gets the number of circuits currently defined.

dss_num_classes() → int
Gets the number of DSS intrinsic classes.

dss_num_user_classes() → int
Gets the number of user-defined classes.

dss_read_allow_forms()
Gets if the DSS allows forms (1) or not (0), default (1).

dss_reset()
Resets DSS initialization for restarts, etc. from applets. Nothing implemented in the OpenDSS Original Source Code

dss_show_panel()
Shows non-MDI child form of the Main DSS Edit form.

dss_start() → int
Validates the user and starts the DSS. Returns TRUE (1) if successful.

dss_write_allow_forms(*argument: int*)
Sets if the DSS allows forms (1) or not (0), default (1). PAY ATTENTION: If arg=0 Then NoFormsAllowed := TRUE (Only set to False) else NoFormsAllowed := FALSE;

class py_dss_interface.models.DSSInterface.DSSInterface.**DSSInterfaceS**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: CStr DSSS(int32_t Parameter, CStr Argument);

This interface returns a string with the result of the query according to the value of the variable Parameter, which can be one of the following.

dss_default_editor() → str
Gets the path name for the default text editor.

dss_new_circuit(*argument: str*) → str
Makes a new circuit, the name of the circuit must be specified in the Argument.

dss_read_datapath() → str
Gets the Data File Path. Default for reports, etc. from DSS.

dss_version() → str
Gets the version string for the DSS.

dss_write_datapath(*argument: str*) → str
Sets the Data File Path. Default for reports, etc. from DSS.

class py_dss_interface.models.DSSInterface.DSSInterface.**DSSInterfaceV**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: void DSSV(int32_t Parameter, VARIANT *Argument);

This interface returns a Variant with the result of the query according to the value of the variable Parameter, which can be one of the following.

dss_classes() → List[str]
Gets the list of DSS intrinsic classes (names of the classes).

dss_user_classes() → List[str]
Gets list of user-defined classes (names of the classes).

4.1.12 DSSProgress

Created by eniocc at 11/10/2020

class py_dss_interface.models.DSSProgress.DSSProgress.**DSSProgress**(*obj_dss*)
Bases: py_dss_interface.models.DSSProgress.DSSProgressI.DSSProgressI, py_dss_interface.models.DSSProgress.DSSProgressS.DSSProgressS

This interface implements the DSSProgress (IDSSProgress) interface of OpenDSS by declaring 2 procedures for accessing the different properties included in this interface: DSSProgressI, DSSProgressS

class py_dss_interface.models.DSSProgress.DSSProgress.**DSSProgressI**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: int32_t DSSProgressI(int32_t Parameter, int32_t Argument);

This interface returns an integer with the result of the query according to the value of the variable Parameter, which can be one of the following.

dssprogress_close() → int
Closes (hides) DSS Progress form.

dssprogress_pct_progress(*arg: float*) → int
Sets the percent progress to indicate [0..100].

dssprogress_show() → int
Shows progress form with null caption and progress set to zero.

class py_dss_interface.models.DSSProgress.DSSProgress.**DSSProgressS**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

TThis interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: CStr DSSProgressS(int32_t Parameter, CStr Argument);

This interface returns a string with the result of the query according to the value of the variable Parameter, which can be one of the following.

dssprogress_caption(*arg: str*) → str
Sets the caption to appear on the bottom of the DSS Progress form.

4.1.13 DSSProperties

Created by eniocc at 11/10/2020

class py_dss_interface.models.DSSProperties.DSSProperties.**DSSProperties**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface implements the DSSproperties (IDSSProperties) interface of OpenDSS by declaring 4 procedures for accessing the different properties included in this interface.

This interface can be used to read/write certain properties of DSS objects.

The structure of the interface is as follows: CStr DSSProperties(int32_t Parameter, CStr Argument);

This interface returns a string pointer (ANSI) with the result of the query according to the value of the variable Parameter, which can be one of the following.

dssproperties_description(*argument: str*) → str
This parameter will deliver the description of the active property. This parameter will deliver the name of the active property. The index of the property must be specified in the argument. The index minimum value is 1. This value must be entered as string.

dssproperties_name(*argument: str*) → str
Delivers the name of the active property. The index of the property must be specified in the argument. The index minimum value is 1. This value must be entered as string.

dssproperties_read_value(*argument: str*) → str
This parameter will deliver the value of the active property. This parameter will deliver the name of the active property. The index of the property must be specified in the argument. The index minimum value is 1. This value must be entered as string.

dssproperties_write_value(*argument: str*) → str
This parameter will allow to set the value of the active property. The new value must be specified in the variable "argument" as string. This parameter will deliver the name of the active property. The index of the property must be specified in the argument. The index minimum value is 1. This value must be entered as string.

4.1.14 ErrorInterface

Created by eniocc at 11/05/2021

class py_dss_interface.models.ErrorInterface.ErrorInterface.**ErrorCode**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: int32_t ErrorCode(void);

error_code() → int
"This interface returns an integer with latest error code delivered by OpenDSS.

class py_dss_interface.models.ErrorInterface.ErrorInterface.**ErrorDesc**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: CStr ErrorDesc(void);

This interface returns a string with description of the latest error code delivered by OpenDSS.

error_desc() → str
"This interface returns a string with description of the latest error code delivered by OpenDSS.

4.1.15 Fuses

Created by eniocc at 11/10/2020

class py_dss_interface.models.Fuses.Fuses.**Fuses**(*obj_dss*)
Bases: py_dss_interface.models.Fuses.FusesI.FusesI, py_dss_interface.models.Fuses.FusesS.FusesS, py_dss_interface.models.Fuses.FusesF.FusesF, py_dss_interface.models.Fuses.FusesV.FusesV

" This interface implements the Fuses (IFuses) interface of OpenDSS by declaring 4 procedures for accessing the different properties included in this interface: FusesI, FusesS, FusesF, FusesV

class py_dss_interface.models.Fuses.Fuses.**FusesF**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: double FusesF(int32_t Parameter, double Argument);

This interface returns a floating point number with the result of the query according to the value of the variable Parameter, which can be one of the following.

fuses_read_delay() → float
Gets the fixed delay time in seconds added to the fuse blowing time determined by the TCC curve. Default is 0.

fuses_read_rated_current() → float
Gets the multiplier or actual amps for the TCCcurve object. Defaults to 1.0, Multiply current values of TCC curve by this to get actual amps.

fuses_write_delay(*argument: float*) → float
Sets the fixed delay time in seconds added to the fuse blowing time determined by the TCC curve. Default is 0.

fuses_write_rated_current(*argument: float*) → float
Sets the multiplier or actual amps for the TCCcurve object. Defaults to 1.0, Multiply current values of TCC curve by this to get actual amps.

class py_dss_interface.models.Fuses.Fuses.**FusesI**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: int32_t FusesI(int32_t Parameter, int32_t Argument);

This interface returns an integer number with the result of the query according to the value of the variable Parameter, which can be one of the following.

fuses_close() → int
Closing of fuse.

fuses_count() → int
Returns the number of Fuses objects currently defined in the active circuit.

fuses_first() → int
Sets the first Fuse to be the active Fuse. Returns 0 if none.

fuses_is_blown() → int
Returns the current state of the fuses. TRUE (1) if any on any phase is blown. Else FALSE (0).

fuses_next() → int
Sets the next Fuse to be the active Fuse. Returns 0 if none.

fuses_num_phases() → int
Gets the number of phases of the active fuse.

fuses_open() → int
Opening of fuse.

fuses_read_idx() → int
Gets the active fuse by index into the list of fuses. 1 based: 1..count.

fuses_read_monitored_term() → int
Gets the terminal number to switch the fuse is connected.

fuses_read_switched_term() → int
Gets the terminal number of the terminal containing the switch controlled by the fuse.

fuses_reset()
Resets the state of the fuse object to the normal state.

fuses_write_idx(*argument: int*) → int
Sets the active fuse by index into the list of fuses. 1 based: 1..count.

fuses_write_monitored_term(*argument: int*) → int
Sets the terminal number to switch the fuse is connected.

fuses_write_switched_term(*argument: int*) → int
Sets the terminal number of the terminal containing the switch controlled by the fuse.

class py_dss_interface.models.Fuses.Fuses.**FusesS**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: CStr FusesS(int32_t Parameter, CStr Argument);

This interface returns a string with the result of the query according to the value of the variable Parameter, which can be one of the following.

fuses_read_monitored_obj() → str
Gets the name of the Monitored Object by the active fuse.

fuses_read_name() → str
Gets the name of the active fuse.

fuses_read_switched_obj() → str
Gets the full name of the circuit element switch that the fuse controls. Defaults to the MonitoredObj.

fuses_read_tcc_curve() → str
Gets the name of the TCCcurve object that determines fuse blowing.

fuses_write_monitored_obj(*argument: str*) → str
Sets the name of the Monitored Object by the active fuse.

fuses_write_name(*argument: str*) → str
Sets the name of the active fuse.

fuses_write_switched_obj(*argument*) → str
Sets the full name of the circuit element switch that the fuse controls. Defaults to the MonitoredObj.

fuses_write_tcc_curve(*argument: str*) → str
Sets the name of the TCCcurve object that determines fuse blowing.

class py_dss_interface.models.Fuses.Fuses.**FusesV**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: void FusesV(int32_t Parameter, VARIANT *Argument);

This interface returns a Variant with the result of the query according to the value of the variable Parameter, which can be one of the following.

fuses_all_names() → List[str]
Gets the variant array of string containing names of all fuses in the circuit.

fuses_read_normal()
Gets a variant array of strings[0..Nphases-1] indicating the normal state for all phases of the active fuse. If value is -1 an error has occurred.

fuses_read_state()
Gets a variant array of strings[0..Nphases-1] indicating the present state for all phases of the active fuse. If value is -1 an error has occurred.

fuses_write_normal(*argument*)
Sets a variant array of strings [0..Nphases-1] indicating the state for all phases of the active fuse. If value is -1 an error has occurred.

fuses_write_state(*argument*)
Sets a variant array of strings [0..Nphases-1] indicating the state for all phases of the active fuse. If value is -1 an error has occurred.

4.1.16 Generators

Created by eniocc at 11/10/2020

class py_dss_interface.models.Generators.Generators.**Generators**(*obj_dss*)
Bases: py_dss_interface.models.Generators.GeneratorsI.GeneratorsI, py_dss_interface.models.Generators.GeneratorsF.GeneratorsF, py_dss_interface.models.Generators.GeneratorsS.GeneratorsS, py_dss_interface.models.Generators.GeneratorsV.GeneratorsV

This interface implements the Generators (IGenerators) interface of OpenDSS by declaring 4 procedures for accessing the different properties included in this interface: GeneratorsI, GeneratorsF, GeneratorsS, GeneratorsV.

class py_dss_interface.models.Generators.Generators.**GeneratorsF**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: double GeneratorsF(int32_t Parameter, double Argument);

This interface returns a floating point number with the result of the query according to the value of the variable Parameter, which can be one of the following.

generators_read_kv() → float
Gets the voltage base for the active generator, kV.

generators_read_kva_rated() → float
Gets the KVA rating of the generator.

generators_read_kvar() → float
Gets the kvar output for the active generator, kW is updated for current power factor.

generators_read_kw() → float
Gets the kW output for the active generator, kvar is updated for current power factor.

generators_read_pf() → float
Gets the power factor (pos. = producing vars). Updates kvar based on present kW value.

generators_read_vmax_pu() → float
Gets the Vmaxpu for Generator Model.

generators_read_vmin_pu() → float
Gets the Vminpu for Generator Model.

generators_write_kv(*argument: float*) → float
Sets the voltage base for the active generator, kV.

generators_write_kva_rated(*argument: float*) → float
Sets the KVA rating of the generator.

generators_write_kvar(*argument: float*) → float
Sets the kvar output for the active generator, kW is updated for current power factor.

generators_write_kw(*argument: float*) → float
Sets the kW output for the active generator, kvar is updated for current power factor.

generators_write_pf(*argument: float*) → float
Sets the power factor (pos. = producing vars). Updates kvar based on present kW value.

generators_write_vmax_pu(*argument: float*) → float
Sets the Vmaxpu for Generator Model.

generators_write_vmin_pu(*argument: float*) → float
Sets the Vminpu for Generator Model.

class py_dss_interface.models.Generators.Generators.**GeneratorsI**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: int32_t GeneratorsI(int32_t Parameter, int32_t Argument);

This interface returns an integer with the result of the query according to the value of the variable Parameter, which can be one of the following.

generators_count() → int
Returns the number of generators Objects in Active Circuit.

generators_first() → int
Sets first generator to be active. Returns 0 if None.

generators_next() → int
Sets next generator to be active. Returns 0 if None.

generators_read_forced_on() → int
Returns 1 if the generator is forced ON regardless of other dispatch criteria; otherwise, returns 0.

generators_read_idx() → int
Gets the active generator by Index into generators list. 1..Count.

generators_read_model() → int
Gets the active generator Model (see Manual for details). 1:Generator injects a constant kW at specified power factor. 2:Generator is modeled as a constant admittance. 3:Const kW, constant kV. Somewhat like a

conventional transmission power flow P-V generator. 4:Const kW, Fixed Q (Q never varies) 5:Const kW, Fixed Q(as a constant reactance) 6:Compute load injection from User-written Model.(see usage of Xd, Xdp) 7:Constant kW, kvar, but current-limited below Vminpu. Approximates a simple inverter. See also Balanced.

generators_read_phases() → int
Returns the number of phases of the active generator.

generators_write_forced_on(*argument: int*) → int
Allows to force ON regardless of other dispatch criteria. To force ON put 1 in the argument, otherwise put 0.

generators_write_idx(*argument: int*) → int
Sets the active generator (argument) by Index into generators list. 1..Count.

generators_write_model(*argument: int*) → int
Sets the active generator Model (see Manual for details). 1:Generator injects a constant kW at specified power factor. 2:Generator is modeled as a constant admittance. 3:Const kW, constant kV. Somewhat like a conventional transmission power flow P-V generator. 4:Const kW, Fixed Q (Q never varies) 5:Const kW, Fixed Q(as a constant reactance) 6:Compute load injection from User-written Model.(see usage of Xd, Xdp) 7:Constant kW, kvar, but current-limited below Vminpu. Approximates a simple inverter. See also Balanced.

generators_write_phases(*argument: int*) → int
Sets the number of phases (argument) of the active generator.

class py_dss_interface.models.Generators.Generators.**GeneratorsS**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: CStr GeneratorsS(int32_t Parameter, CStr Argument);

This interface returns a string as a result of the query according to the value of the variable Parameter, which can be one of the following.

generators_read_name() → str
Gets the name of the active Generator.

generators_write_name(*argument: str*) → str
Sets the name of the active Generator.

class py_dss_interface.models.Generators.Generators.**GeneratorsV**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: void GeneratorsV(int32_t Parameter, VARIANT *Argument);

This interface returns a Variant as a result of the query according to the value of the variable Parameter, which can be one of the following.

generators_all_names() → List[str]
Gets the array of names of all Generator objects.

generators_register_names() → List[str]
Gets the array of names of all generator Energy Meter registers.

generators_register_values() → List[float]
Gets the array of values in generator Energy Meter registers.

4.1.17 ISources

Created by eniocc at 11/10/2020

class py_dss_interface.models.ISources.ISources.**ISources**(*obj_dss*)
Bases: py_dss_interface.models.ISources.ISourcesI.ISourcesI, py_dss_interface.models.ISources.ISourcesF.ISourcesF, py_dss_interface.models.ISources.ISourcesS.ISourcesS, py_dss_interface.models.ISources.ISourcesV.ISourcesV

This interface implements the ISources (IIsources) interface of OpenDSS by declaring 4 procedures for accessing the different properties included in this interface: ISourcesI, ISourcesF, ISourcesS, ISourcesV.

class py_dss_interface.models.ISources.ISources.**ISourcesF**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: double IsourcesF(int32_t Parameter, double Argument);

This interface returns a floating point number with the result of the query according to the value of the variable Parameter, which can be one of the following.

isources_read_amps() → float
Gets the magnitude of the Isource in Amps.

isources_read_angle_deg() → float
Gets the phase angle of the Isource in degrees.

isources_read_frequency() → float
Gets the frequency of the Isource in Hz.

isources_write_amps(*argument: float*) → float
Sets the magnitude of the Isource in Amps.

isources_write_angle_deg(*argument: float*) → float
Sets the phase angle of the Isource in degrees.

isources_write_frequency(*argument: float*) → float
Sets the frequency of the Isource in Hz.

class py_dss_interface.models.ISources.ISources.**ISourcesI**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: int32_t IsourcesI(int32_t Parameter, int32_t Argument);

This interface returns an integer number with the result of the query according to the value of the variable Parameter, which can be one of the following.

isources_count() → int
Returns the number of Isource objects currently defined in the active circuit.

isources_first() → int
Sets the first ISource to be active; returns 0 if none.

isources_next() → int
Sets the next ISource to be active; returns 0 if none.

class py_dss_interface.models.ISources.ISources.**ISourcesS**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: CStr IsourcesS(int32_t Parameter, CStr Argument);

This interface returns a string with the result of the query according to the value of the variable Parameter, which can be one of the following.

isources_read_name() → str
Gets the name of the active Isource object.

isources_write_name(*argument: str*) → str
Sets the name of the active Isource object.

class py_dss_interface.models.ISources.ISources.**ISourcesV**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: void IsourcesV(int32_t Parameter, VARIANT *Argument);

This interface returns a Variant with the result of the query according to the value of the variable Parameter, which can be one of the following.

isources_all_names() → List[str]
Gets the variant array of string containing names of all ISources in the circuit.

4.1.18 LineCodes

Created by eniocc at 11/10/2020

class py_dss_interface.models.LineCodes.LineCodes.**LineCodes**(*obj_dss*)
Bases: py_dss_interface.models.LineCodes.LineCodesF.LineCodesF, py_dss_interface.models.LineCodes.LineCodesS.LineCodesS, py_dss_interface.models.LineCodes.LineCodesI.LineCodesI, py_dss_interface.models.LineCodes.LineCodesV.LineCodesV

This interface implements the Lines (ILineCodes) interface of OpenDSS by declaring 4 procedures for accessing the different properties included in this interface:

class py_dss_interface.models.LineCodes.LineCodes.**LineCodesF**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/modify the properties of the LineCode Class where the values are doubles.

The structure of the interface is as follows: double LineCodesF(int32_t Parameter, double argument)

This interface returns a floating point number, the variable "parameter" is used to specify the property of the class to be used and the variable "argument" can be used to modify the value of the property when necessary. Reading and writing properties are separated and require a different parameter number to be executed.

The properties (parameter) are integer numbers and are described as follows.

linecodes_read_c0() → float
Gets the Zero-sequence capacitance in ohms per unit length for the active LineCode.

linecodes_read_c1() → float
Gets the Positive-sequence capacitance in nF per unit length for the active LineCode.

linecodes_read_emerg_amps() → float
Gets the Emergency ampere rating for the active LineCode.

linecodes_read_norm_amps() → float
Gets the normal ampere rating for the active LineCode.

linecodes_read_r0() → float
Gets the Zero-sequence resistance in ohms per unit length for the active LineCode.

linecodes_read_r1() → float
Gets the Positive-sequence resistance in ohms per unit length for the active LineCode.

linecodes_read_x0() → float
Gets the Zero-sequence reactance in ohms per unit length for the active LineCode.

linecodes_read_x1() → float
Gets the Positive-sequence reactance in ohms per unit length for the active LineCode.

linecodes_write_c0(*argument: float*) → float
Sets the Zero-sequence capacitance in ohms per unit length for the active LineCode. This value must be specified in the argument as a double.

linecodes_write_c1(*argument: float*) → float
Sets the Positive-sequence capacitance in nF per unit length for the active LineCode. This value must be specified in the argument as a double.

linecodes_write_emerg_amps(*argument: float*) → float
Sets the Emergency ampere rating for the active LineCode. This value must be specified in the argument as a double.

linecodes_write_norm_amps(*argument: float*) → float
Sets the normal ampere rating for the active LineCode. This value must be specified in the argument as a double.

linecodes_write_r0(*argument: float*) → float
Sets the Zero-sequence resistance in ohms per unit length for the active LineCode. This value must be specified in the argument as a double.

linecodes_write_r1(*argument: float*) → float
Sets the Positive-sequence resistance in ohms per unit length for the active LineCode. This value must be specified in the argument as a double.

linecodes_write_x0(*argument: float*) → float
Sets the Zero-sequence reactance in ohms per unit length for the active LineCode. This value must be specified in the argument as a double.

linecodes_write_x1(*argument: float*) → float
Sets the Positive-sequence reactance in ohms per unit length for the active LineCode. This value must be specified in the argument as a double.

class py_dss_interface.models.LineCodes.LineCodes.**LineCodesI**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface implements the Lines (ILineCodes) interface of OpenDSS by declaring 4 procedures for accessing the different properties included in this interface:

linecodes_count() → int
Gets the number of Line Objects in Active Circuit.

linecodes_first() → int
Sets the first element active. Returns 0 if no lines. Otherwise, index of the line element.

linecodes_is_z1z0() → int
Gets the flag (Boolean 1/0) denoting whether the impedance data were entered in symmetrical components.

linecodes_next() → int
Sets the next element active. Returns 0 if no lines. Otherwise, index of the line element.

linecodes_read_phases() → int
Delivers the number of phases of the active LineCode as an integer.

linecodes_read_units() → int
Delivers the units of the active LineCode as an integer.

linecodes_write_phases(*argument*) → int
Sets the number of phases of the active LineCode. The units must be specified as an integer in the argument.

linecodes_write_units(*argument: int*) → int
Sets the units of the active LineCode. The units must be specified as an integer in the argument. Please refer to the OpenDSS User manual for more information. UNITS_MAXNUM =9; UNITS_NONE =0; UNITS_MILES =1; UNITS_KFT =2; UNITS_KM =3; UNITS_M =4; UNITS_FT =5; UNITS_IN =6; UNITS_CM =7; UNITS_MM =8;

class py_dss_interface.models.LineCodes.LineCodes.**LineCodesS**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/modify the properties of the LineCode Class where the values are Strings.

The structure of the interface is as follows: CStr LineCodesS(int32_t Parameter, CStr argument)

This interface returns a string, the variable "parameter" is used to specify the property of the class to be used and the variable "argument" can be used to modify the value of the property when necessary. Reading and writing properties are separated and require a different parameter number to be executed.

The properties (parameter) are integer numbers and are described as follows.;

linecodes_read_name() → str
Gets the name of the active LineCode element.

linecodes_write_name(*argument: str*) → str
Sets the name of the active LineCode element. The new value must be specified in the argument as a string.

class py_dss_interface.models.LineCodes.LineCodes.**LineCodesV**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/modify the properties of the LineCode Class where the values are Variants.

The structure of the interface is as follows: void LineCodesV(int32_t Parameter, , VARIANT *Argument);

This interface returns a Variant, the variable "parameter" is used to specify the property of the class to be used and the variable "argument" can be used to modify the value of the property when necessary. Reading and writing properties are separated and require a different parameter number to be executed.

The properties (parameter) are integer numbers and are described as follows.

linecodes_all_names() → str
Gets the capacitance matrix in ohms per unit length of the active LineCode.

linecodes_read_cmatrix() → str
Gets the capacitance matrix in ohms per unit length of the active LineCode.

linecodes_read_rmatrix() → str
Gets the resistance matrix in ohms per unit length of the active LineCode.

linecodes_read_xmatrix() → str
Gets the reactance matrix in ohms per unit length of the active LineCode.

linecodes_write_cmatrix(*argument: str*) → None
Sets the capacitance matrix in ohms per unit length of the active LineCode. The new values must be entered as a vector of doubles using the argument. :param argument: must be a string like that [383.948 |0 383.948 |0 0 383.948]

linecodes_write_rmatrix (*argument: str*) → None
Sets the resistance matrix in ohms per unit length of the active LineCode. The new values must be entered as a vector of doubles using the argument. :param argument: must be a string like that [0.791721 | 0.318476 0.781649 | 0.28345, 0.318476, 0.791721]

linecodes_write_xmatrix (*argument: str*) → None
Sets the reactance matrix in ohms per unit length of the active LineCode. The new values must be entered as a vector of doubles using the argument.

4.1.19 Lines

Created by eniocc at 11/10/2020

class py_dss_interface.models.Lines.Lines.**Lines** (*obj_dss*)
Bases: py_dss_interface.models.Lines.LinesV.LinesV, py_dss_interface.models.Lines.LinesS.LinesS, py_dss_interface.models.Lines.LinesI.LinesI, py_dss_interface.models.Lines.LinesF.LinesF

This interface implements the Lines (ILines) interface of OpenDSS by declaring 4 procedures for accessing the different properties included in this interface: LinesV, LinesS, LinesI, LinesF.

class py_dss_interface.models.Lines.Lines.**LinesF** (*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/modify the properties of the Lines Class where the values are doubles.

The structure of the interface is as follows: double LinesF(int32_t Parameter, double argument)

This interface returns a floating point number, the variable "parameter" is used to specify the property of the class to be used and the variable "argument" can be used to modify the value of the property when necessary. Reading and writing properties are separated and require a different parameter number to be executed.

The properties (parameter) are integer numbers and are described as follows.

lines_read_c0 () → float
Gets the zero sequence capacitance, nanofarads per unit length.

lines_read_c1 () → float
Gets the positive sequence capacitance, nanofarads per unit length.

lines_read_emerg_amps () → float
Gets the emergency (maximum) ampere rating of Line.

lines_read_length () → float
Gets the length of line section in units compatible with the LineCode definition.

lines_read_norm_amps () → float
Gets the normal ampere rating of line section.

lines_read_r0 () → float
Gets the zero sequence resistance, ohm per unit length.

lines_read_r1 () → float
Gets the positive sequence resistance, ohm per unit length.

lines_read_rg () → float
Gets the earth return value used to compute line impedance's at power frequency.

lines_read_rho () → float
Gets the earth resistivity, m-ohms.

lines_read_season_rating() → float
Returns the rating for the current season (in Amps) if the SeasonalRatings option is active.

lines_read_x0() → float
Gets the zero sequence reactance, ohm per unit length.

lines_read_x1() → float
Gets the positive sequence reactance, ohm per unit length.

lines_read_xg() → float
Gets the earth return reactance value used to compute line impedances at power frequency.

lines_write_c0(*argument: float*) → float
Sets the zero sequence capacitance, nanofarads per unit length.

lines_write_c1(*argument: float*) → float
Sets the positive sequence capacitance, nanofarads per unit length.

lines_write_emerg_amps(*argument: float*) → float
Sets the emergency (maximum) ampere rating of Line.

lines_write_length(*argument: float*) → float
Sets the length of line section in units compatible with the LineCode definition.

lines_write_norm_amps(*argument: float*) → float
Sets the normal ampere rating of Line.

lines_write_r0(*argument: float*) → float
Sets the zero sequence resistance, ohm per unit length.

lines_write_r1(*argument: float*) → float
Sets the positive sequence resistance, ohm per unit length.

lines_write_rg(*argument: float*) → float
Sets the earth return value used to compute line impedances at power frequency.

lines_write_rho(*argument: float*) → float
Sets the earth resistivity, m-ohms.

lines_write_x0(*argument: float*) → float
Sets the zero sequence reactance, ohm per unit length.

lines_write_x1(*argument: float*) → float
Sets the positive sequence reactance, ohm per unit length.

lines_write_xg(*argument: float*) → float
Sets the earth return reactance value used to compute line impedances at power frequency.

class py_dss_interface.models.Lines.Lines.**LinesI**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/modify the properties of the Lines Class where the values are integers.

The structure of the interface is as follows: int32_t LinesI(int32_t Parameter, int32_t argument)

This interface returns an integer (signed 32 bits), the variable "parameter" is used to specify the property of the class to be used and the variable "argument" can be used to modify the value of the property when necessary. Reading and writing properties are separated and require a different parameter number to be executed.

The properties (parameter) are integer numbers and are described as follows.

lines_count() → int
Gets the number of Line Objects in Active Circuit.

lines_first() → int
Sets the first element active. Returns 0 if no lines. Otherwise, index of the line element.

lines_next() → int
Sets the next element active. Returns 0 if no lines. Otherwise, index of the line element.

lines_num_cust() → int
Gets the number of customers on this line section.

lines_parent() → int
Gets the parents of the active Line to be the active Line. Return 0 if no parent or action fails.

lines_read_phases() → int
Gets the number of phases of the active line object.

lines_read_units() → int
Gets the units of the line (distance, check manual for details).

lines_write_phases(*argument: int*) → int
Sets the number of phases of the active line object.

lines_write_units(*argument: int*) → int
Sets the units of the line (distance, check manual for details). units: {none | mi|kft|km|m|Ft|in|cm } UNITS_MAXNUM =9; UNITS_NONE =0; UNITS_MILES =1; UNITS_KFT =2; UNITS_KM =3; UNITS_M =4; UNITS_FT =5; UNITS_IN =6; UNITS_CM =7; UNITS_MM =8;

class py_dss_interface.models.Lines.Lines.**LinesS**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/modify the properties of the Lines Class where the values are Strings.

The structure of the interface is as follows: CStr LinesS(int32_t Parameter, CStr argument)

This interface returns a string, the variable "parameter" is used to specify the property of the class to be used and the variable "argument" can be used to modify the value of the property when necessary. Reading and writing properties are separated and require a different parameter number to be executed.

The properties (parameter) are integer numbers and are described as follows.

lines_read_bus1() → str
Gets the name of bus for terminal 1.

lines_read_bus2() → str
Gets the name of bus for terminal 2.

lines_read_geometry() → str
Gets the name of the Line geometry code.

lines_read_linecode() → str
Gets the name of LineCode object that defines the impedances.

lines_read_name() → str
Gets the name of the active Line element.

lines_read_spacing() → str
Gets the name of the Line spacing code.

lines_write_bus1(*argument: str*) → str
Sets the name of bus for terminal 1.

lines_write_bus2(*argument: str*) → str
Sets the name of bus for terminal 2.

lines_write_geometry(*argument: str*) → str
Sets the name of the Line geometry code.

lines_write_linecode(*argument: str*) → str
Sets the name of LineCode object that defines the impedances.

lines_write_name(*argument: str*) → str
Sets the name of the Line element to set it active.

lines_write_spacing(*argument: str*) → str
Sets the name of the Line spacing code.

class py_dss_interface.models.Lines.Lines.**LinesV**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/modify the properties of the Lines Class where the values are Variants.

The structure of the interface is as follows: void LinesV(int32_t Parameter, , VARIANT *Argument);

This interface returns a Variant, the variable "parameter" is used to specify the property of the class to be used and the variable "argument" can be used to modify the value of the property when necessary. Reading and writing properties are separated and require a different parameter number to be executed.

The properties (parameter) are integer numbers and are described as follows.

lines_all_names() → str
Gets the name of all Line Objects.

lines_read_cmatrix() → str
Gets the capacitance matrix (full), nanofarads per unit length. Variant array of doubles.

lines_read_rmatrix()
Gets the resistance matrix (full), ohms per unit length. Variant array of doubles.

lines_read_xmatrix() → List[float]
Gets the reactance matrix (full), ohms per unit length. Variant array of doubles.

lines_read_yprim() → str
Gets the YPrimitive of the active Line.

lines_write_cmatrix(*argument*) → int
Sets the capacitance matrix (full), nanofarads per unit length. Variant array of doubles.

lines_write_rmatrix(*argument*) → str
Sets the resistance matrix (full), ohms per unit length. Variant array of doubles.

lines_write_xmatrix(*argument*) → str
Sets the reactance matrix (full), ohms per unit length. Variant array of doubles.

lines_write_yprim(*argument*) → str
According the oficial documentation this parameter does nothing at present.

4.1.20 Loads

Created by eniocc at 11/10/2020

class py_dss_interface.models.Loads.Loads.**Loads**(*obj_dss*)
Bases: py_dss_interface.models.Loads.LoadsF.LoadsF, py_dss_interface.models.Loads.LoadsI.LoadsI, py_dss_interface.models.Loads.LoadsS.LoadsS, py_dss_interface.models.Loads.LoadsV.LoadsV

This interface implements the Loads (ILoads) interface of OpenDSS by declaring 4 procedures for accessing the different properties included in this interface: LoadsF, LoadsI, LoadsS, LoadsV.

class py_dss_interface.models.Loads.Loads.**LoadsF**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/modify the properties of the Loads Class where the values are floating point numbers (double).

The structure of the interface is as follows: double DSSLoadsF(int32_t Parameter,double Argument);

This interface returns a Double (IEEE 754 64 bits), the variable "parameter" (Integer) is used to specify the property of the class to be used and the variable "argument" (double) can be used to modify the value of the property when necessary. Reading and writing properties are separated and require a different parameter number to be executed.

The properties (parameter) are integer numbers and are described as follows.

loads_read_allocation_factor() → float
Allows to read the AllocationFactor property of the active load. The parameter argument can be filled with a 0.

loads_read_c_factor() → float
Allows to read the CFactor property of the active load. The parameter argument can be filled with a 0.

loads_read_cvr_vars() → float
Allows to read the CVRvars property of the active load. The parameter argument can be filled with a 0.

loads_read_cvr_watts() → float
Allows to read the CVRWatts property of the active load. The parameter argument can be filled with a 0.

loads_read_kv() → float
Allows to read the kV property of the active load. The parameter argument can be filled with a 0.

loads_read_kva() → float
Allows to read the kva property of the active load. The parameter argument can be filled with a 0.

loads_read_kvar() → float
Allows to read the kvar property of the active load. The parameter argument can be filled with a 0.

loads_read_kw() → float
Allows to read the kW property of the active load. The parameter argument can be filled with a 0.

loads_read_kwh() → float
Allows to read the kWh property of the active load. The parameter argument can be filled with a 0.

loads_read_kwh_days() → float
Allows to read the kWhdays property of the active load. The parameter argument can be filled with a 0.

loads_read_pct_mean() → float
Allows to read the PctMean property of the active load. The parameter argument can be filled with a 0.

loads_read_pct_series_rl() → float
allows to read the PctSeriesRL (Percent of Load that is modeled as series R-L for harmonic studies) property of the active load. The parameter argument can be filled with a 0.

loads_read_pct_std_dev() → float
Allows to read the PctStdDev property of the active load. The parameter argument can be filled with a 0.

loads_read_pf() → float
Allows to read the pf property of the active load. The parameter argument can be filled with a 0.

loads_read_r_neut() → float
Allows to read the RNeut (neutral resistance for wye connected loads) property of the active load. The parameter argument can be filled with a 0.

loads_read_rel_weight() → float
Allows to read the RelWeight (relative weighting factor) property of the active load. The parameter argument can be filled with a 0.

loads_read_vmax_pu() → float
Allows to read the VMaxpu property of the active load. The parameter argument can be filled with a 0.

loads_read_vmin_emerg() → float
Allows to read the VMinemerg property of the active load. The parameter argument can be filled with a 0.

loads_read_vmin_norm() → float
Allows to read the VMinnorm property of the active load. The parameter argument can be filled with a 0.

loads_read_vmin_pu() → float
Allows to read the VMinpu property of the active load. The parameter argument can be filled with a 0.

loads_read_x_neut() → float
Allows to read the Xneut property of the active load. The parameter argument can be filled with a 0.

loads_read_xfkva() → float
Allows to read the xfKVA (Rated service transformer KVA for load allocation, using Allocationfactor. Affects kW, kvar and pf.) property of the active load. The parameter argument can be filled with a 0.

loads_write_allocation_factor(*argument*) → float
Allows to write the AllocationFactor property of the active load. The parameter argument must contain the new value in AllocationFactor for the desired active load. The return value will be equal to 0.

loads_write_c_factor(*argument: float*) → float
Allows to write the CFactor property of the active load. The parameter argument must contain the new value in CFactor for the desired active load. The return value will be equal to 0.

loads_write_cvr_vars(*argument*) → float
Allows to write the CVRvars property of the active load. The parameter argument must contain the new value in CVRWatts for the desired active load. The return value will be equal to 0.

loads_write_cvr_watts(*argument*) → float
Allows to write the CVRWatts property of the active load. The parameter argument must contain the new value in CVRWatts for the desired active load. The return value will be equal to 0.

loads_write_kv(*argument*) → float
Allows to write the kV property of the active load. The parameter argument must contain the new value in kV for the desired active load. The return value will be equal to 0.

loads_write_kva(*argument*) → float
Allows to write the kva property of the active load. The parameter argument must contain the new value in kva for the desired active load. The return value will be equal to 0.

loads_write_kvar(*argument*) → float
Allows to write the kvar property of the active load. The parameter argument must contain the new value in kvar for the desired active load. The return value will be equal to 0.

loads_write_kw(*argument*) → float
Allows to write the kW property of the active load. The parameter argument must contain the new value in kW for the desired active load. The return value will be equal to 0.

loads_write_kwh(*argument*) → float
Allows to write the kWh property of the active load. The parameter argument must contain the new value in kWh for the desired active load. The return value will be equal to 0.

loads_write_kwh_days(*argument*) → float
Allows to write the kWhdays property of the active load. The parameter argument must contain the new value in kWhdays for the desired active load. The return value will be equal to 0.

loads_write_pct_mean(*argument*) → float
Allows to write the PctMean property of the active load. The parameter argument must contain the new value in PctMean for the desired active load. The return value will be equal to 0.

loads_write_pct_series_rl(*argument*) → float
allows to write the PctSeriesRL (Percent of Load that is modeled as series R-L for harmonic studies) property of the active load. The parameter argument must contain the new value in PctSeriesRL for the desired active load. The return value will be equal to 0.

loads_write_pct_std_dev(*argument*) → float
Allows to write the PctStdDev property of the active load. The parameter argument must contain the new value in PctStdDev for the desired active load. The return value will be equal to 0.

loads_write_pf(*argument*) → float
Allows to write the pf property of the active load. The parameter argument must contain the new value in pf for the desired active load. The return value will be equal to 0.

loads_write_r_neut(*argument*) → float
Allows to write the RNeut (neutral resistance for wye connected loads) property of the active load. The parameter argument must contain the new value in RNeut for the desired active load. The return value will be equal to 0.

loads_write_rel_weight(*argument*) → float
Allows to write the RelWeight (relative weighting factor) property of the active load. The parameter argument must contain the new value in RelWeight for the desired active load. The return value will be equal to 0.

loads_write_vmax_pu(*argument*) → float
Allows to write the VMaxpu property of the active load. The parameter argument must contain the new value in VMaxpu for the desired active load. The return value will be equal to 0.

loads_write_vmin_emerg(*argument*) → float
Allows to write the VMinemerg property of the active load. The parameter argument must contain the new value in VMinemerg for the desired active load. The return value will be equal to 0.

loads_write_vmin_norm(*argument*) → float
Allows to write the VMinnorm property of the active load. The parameter argument must contain the new value in VMinnorm for the desired active load. The return value will be equal to 0.

loads_write_vmin_pu(*argument*) → float
Allows to write the VMinpu property of the active load. The parameter argument must contain the new value in VMinpu for the desired active load. The return value will be equal to 0.

loads_write_x_neut(*argument*) → float
Allows to write the Xneut property of the active load. The parameter argument must contain the new value in Xneut for the desired active load. The return value will be equal to 0.

loads_write_xfkva(*argument*) → float
Allows to write the xfKVA (Rated service transformer KVA for load allocation, using Allocationfactor. Affects kW, kvar and pf.) property of the active load. The parameter argument must contain the new value in xfKVA for the desired active load. The return value will be equal to 0.

class py_dss_interface.models.Loads.Loads.**LoadsI**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/modify the properties of the Loads Class where the values are integers.

The structure of the interface is as follows: int32_t DSSLoads(int32_t Parameter, int32_t argument)

This interface returns an integer (signed 32 bits), the variable "parameter" is used to specify the property of the class to be used and the variable "argument" can be used to modify the value of the property when necessary.

Reading and writing properties are separated and require a different parameter number to be executed.

The properties (parameter) are integer numbers and are described as follows.

loads_count() → int
Returns the number of load elements within the active circuit. The parameter argument can be filled with a 0.

loads_first() → int
Allows to set the active load into the first load registered in the active circuit. As a result, this property will return the number 1. The parameter argument can be filled with a 0.

loads_next() → int
Sets the active load into the next load registered in the active circuit. As a result, this property will deliver the index of the active load. The parameter argument can be filled with a 0.

loads_read_class() → int
Allows to read the code number used to separate loads by class or group. The parameter argument can be filled with a 0.

loads_read_idx() → int
Allows to read the index of the active load. The parameter argument can be filled with a 0.

loads_read_is_delta() → int
Allows to read if the active load is connected in delta, if the answer is positive, this function will deliver a 1; otherwise, the answer will be 0. The parameter argument can be filled with a 0.

loads_read_model() → int
Allows to read the model of the active load. The parameter argument can be filled with a 0.

loads_read_num_cust() → int
Allows to read the number of customer of the active load. The parameter argument can be filled with a 0.

loads_read_status() → int
Allows to read Response to load multipliers: Fixed (growth only - 1), Exempt (no LD curve - 2), Variable (all - 0), of the active load. The parameter argument can be filled with a 0.

loads_write_class(*argument*) → int
Allows to read the code number used to separate loads by class or group. The parameter argument can be filled with a 0.

loads_write_idx(*argument*) → int
Allows to write the index of the active load. The parameter argument must contain the index of the desired active load. The return value will be equal to 0.

loads_write_is_delta(*argument*) → int
Allows to read if the active load is connected in delta, if the answer is positive, this function will deliver a 1; otherwise, the answer will be 0. This parameter will return a 0.

loads_write_model(*argument*) → int
Allows to write the model of the active load using the parameter argument. This parameter will return a 0.

loads_write_num_cust(*argument*) → int
Allows to write the number of customers of the active load using the parameter argument. This parameter will return a 0.

loads_write_status(*argument*) → int
Allows to read Response to load multipliers: Fixed (growth only - 1), Exempt (no LD curve - 2), Variable (all - 0), of the active load. This parameter will return a 0.

class py_dss_interface.models.Loads.Loads.**LoadsS**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/modify the properties of the Loads Class where the values are strings.

The structure of the interface is as follows: CStr DSSLoadsS(int32_t Parameter, CStr Argument);

This interface returns a string, the variable "parameter" (Integer) is used to specify the property of the class to be used and the variable "argument" (string) can be used to modify the value of the property when necessary. Reading and writing properties are separated and require a different parameter number to be executed.

loads_read_cvr_curve() → str
Allows to read the CVRCurve property of the active load. The parameter argument can be filled with an empty string.

loads_read_daily() → str
Allows to read the daily property of the active load. The parameter argument can be filled with an empty string.

loads_read_duty() → str
Allows to read the duty property of the active load. The parameter argument can be filled with an empty string.

loads_read_growth() → str
Allows to read the Growth property of the active load. The parameter argument can be filled with an empty string.

loads_read_name() → str
Allows to read the Name property of the active load. The parameter argument can be filled with an empty string.

loads_read_spectrum() → str
Allows to read the Spectrum property of the active load. The parameter argument can be filled with an empty string.

loads_read_yearly() → str
Allows to read the Yearly property of the active load. The parameter argument can be filled with an empty string.

loads_write_cvr_curve(*argument*) → str
Allows to set the CVRCurve property for the active load. The parameter argument must contain the Name of the new CVRCurve to be linked to the active load. The return value will be equal to empty.

loads_write_daily(*argument*) → str
Allows to set the daily property for the active load. The parameter argument must contain the Name of the new daily to be linked to the active load. The return value will be equal to empty.

loads_write_duty(*argument*) → str
Allows to set the dduty property for the active load. The parameter argument must contain the Name of the new duty to be linked to the active load. The return value will be equal to empty.

loads_write_growth(*argument*) → str
Allows to set the Growth property for the active load. The parameter argument must contain the Name of the new Growth to be linked to the active load. The return value will be equal to empty.

loads_write_name(*argument*) → str
allows to set the active load by specifying the Name load. The parameter argument must contain the Name of the load to activate. The return value will be equal to empty.

loads_write_spectrum(*argument*) → str
Allows to set the Spectrum property for the active load. The parameter argument must contain the Name of the new Spectrum to be linked to the active load. The return value will be equal to empty.

loads_write_yearly(*argument*) → str
Allows to set the Yearly property for the active load. The parameter argument must contain the Name of the new Yearly to be linked to the active load. The return value will be equal to empty.

class py_dss_interface.models.Loads.Loads.**LoadsV**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/modify the properties of the Loads Class where the values are variants (the value can have different formats).

The structure of the interface is as follows: void DSSLoadsV(int32_t Parameter, VARIANT *Argument);

This interface returns a string, the variable "parameter" (Integer) is used to specify the property of the class to be used and the variable "argument" (Variant) is used to return the variant structure.

loads_all_names()
Allows to read the names of all the loads present in the active circuit. The result is delivered as variant, however, the content of this variant is an array of strings.

loads_read_zipv()
Allows to read the array of 7 elements (doubles) for ZIP property of the active Load object.

loads_write_zipv(*argument*)
Allows to write the array of 7 elements (doubles) for ZIP property of the active Load object. :param argument: Array of 7 coefficients:

First 3 are ZIP weighting factors for real power (should sum to 1) Next 3 are ZIP weighting factors for reactive power (should sum to 1) Last 1 is cut-off voltage in p.u. of base kV; load is 0 below this cut-off No defaults; all coefficients must be specified if using model=8.

4.1.21 LoadShapes

Created by eniocc at 11/10/2020

class py_dss_interface.models.LoadShapes.LoadShapes.**LoadShapes**(*obj_dss*)
Bases: py_dss_interface.models.LoadShapes.LoadShapesF.LoadShapesF, py_dss_interface.models.LoadShapes.LoadShapesI.LoadShapesI, py_dss_interface.models.LoadShapes.LoadShapesS.LoadShapesS, py_dss_interface.models.LoadShapes.LoadShapesV.LoadShapesV

This interface implements the LoadShape (ILoadShape) interface of OpenDSS by declaring 4 procedures for accessing the different properties included in this interface: .

class py_dss_interface.models.LoadShapes.LoadShapes.**LoadShapesF**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: double LoadShapeF(int32_t Parameter, double Argument);

This interface returns a floating point number with the result of the query according to the value of the variable Parameter, which can be one of the following.

loadshapes_read_hr_interval() → float
Gets the fixed interval time value, hours.

loadshapes_read_min_interval() → float
Gets the fixed interval time value, in minutes.

loadshapes_read_p_base() → float
Gets the base for normalizing P curve. If left at zero, the peak value is used.

loadshapes_read_q_base() → float
Gets the base for normalizing Q curve. If left at zero, the peak value is used.

loadshapes_read_s_interval() → float
Gets the fixed interval data time interval, seconds.

loadshapes_write_hr_interval(*argument: float*) → float
Sets the fixed interval time value, hours.

loadshapes_write_min_interval(*argument: float*) → float
Sets the fixed interval time value, in minutes.

loadshapes_write_p_base(*argument: float*) → float
Sets the base for normalizing P curve. If left at zero, the peak value is used.

loadshapes_write_q_base(*argument: float*) → float
Sets the base for normalizing Q curve. If left at zero, the peak value is used.

loadshapes_write_s_interval(*argument: float*) → float
Sets the fixed interval data time interval, seconds.

class py_dss_interface.models.LoadShapes.LoadShapes.**LoadShapesI**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: int32_t LoadShapeI(int32_t Parameter, int32_t Argument);

This interface returns an integer number with the result of the query according to the value of the variable Parameter, which can be one of the following.

loadshapes_count() → int
Returns the number of LoadShape objects currently defined in LoadShape collection.

loadshapes_first() → int
sets the first loadshape active and return integer index of the loadshape. Returns 0 if no more.

loadshapes_next() → int
Sets the next loadshape active and return integer index of the loadshape. Returns 0 if no more.

loadshapes_normalize() → int
Normalizes the P and Q curves based on either Pbase, Qbase or simply the peak value of the curve.

loadshapes_read_npts() → int
Gets the number of points in active LoadShape.

loadshapes_read_use_actual() → int
Gets a TRUE/FALSE (1/0) to let Loads know to use the actual value in the curve rather than use the value as a multiplier.

loadshapes_write_npts(*argument*) → int
Sets the number of points in active LoadShape.

loadshapes_write_use_actual(*argument*) → int
Sets a TRUE/FALSE (1/0 - Argument) to let Loads know to use the actual value in the curve rather than use the value as a multiplier.

class py_dss_interface.models.LoadShapes.LoadShapes.**LoadShapesS**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: CStr LoadShapeS(int32_t Parameter, CStr Argument);

This interface returns string with the result of the query according to the value of the variable Parameter, which can be one of the following.

loadshapes_read_name()
Gets the name of the active LoadShape object.

loadshapes_write_name(*argument*)
Sets the name of the active LoadShape object.

class py_dss_interface.models.LoadShapes.LoadShapes.**LoadShapesV**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: void LoadShapeV(int32_t Parameter, VARIANT *Argument);

This interface returns a Variant with the result of the query according to the value of the variable Parameter, which can be one of the following.

loadshapes_all_names()
Gets a variant array of strings containing names of all LoadShape objects currently defined.

loadshapes_read_p_mult()
Gets a variant array of doubles for the P multiplier in the LoadShape.

loadshapes_read_q_mult()
Gets a variant array of doubles for the Q multiplier in the LoadShape.

loadshapes_read_time_array()
Gets a time array in hours corresponding to P and Q multipliers when the Interval = 0.

loadshapes_write_p_mult(*argument*)
Sets a variant array of doubles for the P multiplier in the LoadShape.

loadshapes_write_q_mult(*argument*)
Sets a variant array of doubles for the Q multiplier in the LoadShape.

loadshapes_write_time_array(*argument*)
Sets a time array in hours corresponding to P and Q multipliers when the Interval = 0.

4.1.22 Meters

Created by eniocc at 11/10/2020

class py_dss_interface.models.Meters.Meters.**Meters**(*obj_dss*)
Bases: py_dss_interface.models.Meters.MetersV.MetersV, py_dss_interface.models.Meters.MetersS.MetersS, py_dss_interface.models.Meters.MetersF.MetersF, py_dss_interface.models.Meters.MetersI.MetersI

This interface implements the Meters (IMeters) interface of OpenDSS by declaring 4 procedures for accessing the different properties included in this interface: MetersV, MetersS, MetersF, MetersI.

class py_dss_interface.models.Meters.Meters.**MetersF**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: double MetersF(int32_t Parameter, double Argument);

This interface returns a floating point number (64 bits) according to the number sent in the variable "parameter". The parameter can be one of the following.

meters_avg_repair_time() → float
Returns the average Repair Time in this Section of the meter zone.

meters_cust_interrupts() → float
Returns the total customer interruptions for this meter zone based on reliability calcs.

meters_fault_rate_x_repair_hrs() → float
Returns the sum of Fault Rate Time Repair Hours in this section of the meter zone.

meters_saidi() → float
Returns the SAIDI for this meter zone. Execute DoreliabilityCalc first.

meters_saifi() → float
Returns SAIFI for this meter's zone. Execute reliability calc method first.

meters_saifi_kw() → float
Returns the SAIFI based on kW rather than number of customers. Get after reliability calcs.

meters_sum_branch_flt_rates() → float
Returns the sum of the branch fault rates in this section of the meter's zone.

class py_dss_interface.models.Meters.Meters.**MetersI**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: int32_t MetersI(int32_t Parameter, int32_t Argument);

This interface returns an integer according to the number sent in the variable "parameter". The parameter can be one of the following.

meters_close_all_di_files() → int
Closes all Demand Interval (DI) files. Necessary at the end of a run.

meters_count() → int
Returns the number of Energy Meters in the Active Circuit.

meters_count_branches() → int
Returns the number of branches in active Energy Meter zone (same as sequencelist size).

meters_count_end_elements() → int
Returns the number of zone end elements in the active meter zone.

meters_di_files_are_open() → int
Returns a global flag (1=true, 0=false) to indicate if Demand Interval (DI) files have been properly opened.

meters_do_reliability_calc() → int
Calculates SAIFI, etc. if the Argument is equal to 1 this parameter will assume restoration, otherwise it will not.

meters_first() → int
Sets the first Energy Meter active. Returns 0 if no monitors.

meters_next() → int
Sets the next energy Meter Active. Returns 0 if no more.

meters_num_section_branches() → int
Returns the number of branches (lines) in the active section.

meters_num_section_customers() → int
Returns the number of customers in the active section.

meters_num_sections() → int
Returns the number of feeder sections in this meter's zone.

meters_ocp_device_type() → int
Returns the type of OCP device: {1=fuse | 2+ recloser | 3= relay}.

meters_open_all_di_files() → int
Opens Demand Interval (DI) files. Returns 0.

meters_read_metered_terminal() → int
Returns the number of metered terminal by the active Energy Meter.

meters_read_sequence_index() → int
Returns the index into meter's SequenceList that contains branch pointers in lexical order. Earlier index guaranteed to be up line from later index. Sets PDElement active.

meters_reset() → int
Resets the active Meter object.

meters_reset_all() → int
Resets all Meter object.

meters_sample() → int
Causes active meter to take a sample.

meters_sample_all() → int
Causes all Energy Meters to take a sample of the present state. Returns 0.

meters_save() → int
Causes active meter to save its current sample buffer to its meter stream. Then you can access the Bytestream or channel data. Most standard solution modes do this automatically.

meters_save_all() → int
Save all Energy Meter buffers to their respective file streams. Returns 0.

meters_sect_seq_idx() → int
Returns the Sequence Index of the branch at the head of this section.

meters_sect_total_cust() → int
Returns the total customers down line from this section.

meters_seq_list_size() → int
Returns the size of Sequence List.

meters_set_active_section(*argument*) → int
Sets the designated section (argument) if the index is valid.

meters_total_customers() → int
Returns the total number of customers in this zone (down line from the Energy Meter).

meters_write_metered_terminal(*argument*) → int
Sets the number of metered terminal by the active Energy Meter.

meters_write_sequence_index(*argument*) → int
Sets the index into meter's SequenceList that contains branch pointers in lexical order. Earlier index guaranteed to be up line from later index. Sets PDElement active.

class py_dss_interface.models.Meters.Meters.**MetersS**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: CStr MetersS(int32_t Parameter, CStr Argument);

This interface returns a string according to the number sent in the variable "parameter". The parameter can be one of the following.

meters_read_metered_element()
Returns the name of the metered element (considering the active Energy Meter).

meters_read_name()
Returns the active Energy Meter's name.

meters_write_metered_element(*argument*)
Sets the name of the metered element (considering the active Energy Meter).

meters_write_name(*argument*)
Sets the active Energy Meter's name.

class py_dss_interface.models.Meters.Meters.**MetersV**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: void MetersV(int32_t Parameter, VARIANT *Argument);

This interface returns a variant according to the number sent in the variable "parameter". The parameter can be one of the following.

meters_all_branches_in_zone()
Returns a wide string list of all branches in zone of the active Energy Meter object.

meters_all_end_elements()
Returns a variant array of names of all zone end elements.

meters_all_names()
Returns an array of all Energy Meter names.

meters_all_pce_in_zone()
This parameter returns a wide string list of all the PCE in zone of the active Energy Meter object.

meters_read_alloc_factors()
Returns an array of doubles: allocation factors for the active Meter.

meters_read_calc_current()
Returns the magnitude of the real part of the Calculated Current (normally determined by solution) for the meter to force some behavior on Load Allocation.

meters_read_peak_current()
Returns an array of doubles with the Peak Current Property.

meters_register_names()
Returns an array of strings containing the names of the registers.

meters_register_values()
Returns an array of values contained in the Meter registers for the active Meter.

meters_totals()
Returns the totals for all registers of all Meters.

meters_write_alloc_factors(*argument*)
Receives an array of doubles to set the phase allocation factors for the active Meter.

meters_write_calc_current(*argument: str*)
Sets the magnitude of the real part of the Calculated Current (normally determined by solution) for the meter to force some behavior on Load Allocation.

meters_write_peak_current(*argument*)
Receives an array of doubles to set values of Peak Current Property.

4.1.23 Monitors

Created by eniocc at 11/10/2020

class py_dss_interface.models.Monitors.Monitors.**Monitors**(*obj_dss*)
Bases: py_dss_interface.models.Monitors.MonitorsI.MonitorsI, py_dss_interface.models.Monitors.MonitorsV.MonitorsV, py_dss_interface.models.Monitors.MonitorsS.MonitorsS

This interface implements the Monitors (IMonitors) interface of OpenDSS by declaring 3 procedures for accessing the different properties included in this interface: MonitorsI, MonitorsV, MonitorsS.

class py_dss_interface.models.Monitors.Monitors.**MonitorsI**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: int32_t MonitorsI(int32_t Parameter, int32_t Argument);

This interface returns an integer according to the number sent in the variable "parameter". The parameter can be one of the following.

monitors_count() → int
Returns the number of monitors.

monitors_file_version() → int
Returns the Monitor File version (integer).

monitors_first() → int
Sets the first monitor active. Returns 0 if no monitors.

monitors_next() → int
Set the next monitor active. Returns 0 if no more.

monitors_num_channels() → int
Returns the number of Channels on the active Monitor.

monitors_process() → int
Post-process monitor examples taken so far, e.g., Pst for mode = 4.

monitors_process_all() → int
Makes that all monitors post-process the data taken so far.

monitors_read_mode() → int
Returns the monitor mode (bitmask integer - see DSS Help).

monitors_read_terminal() → int
Returns the terminal number of element being monitored.

monitors_record_size() → int
Returns the size of each record in ByteStream.

monitors_reset() → int
Resets the active Monitor object.

monitors_reset_all() → int
Resets all Monitor object.

monitors_sample() → int
Causes active monitor to take a sample.

monitors_sample_all() → int
Causes all Monitors to take a sample of the present state. Returns 0.

monitors_sample_count() → int
Returns number of examples in Monitor at present.

monitors_save() → int
Causes active monitor to save its current sample buffer to its monitor stream. Then you can access the Bytestream or channel data. Most standard solution modes do this automatically.

monitors_save_all() → int
Save all Monitor buffers to their respective file streams. Returns 0.

monitors_show() → int
Converts monitor file into text and displays with text editor.

monitors_write_mode(*argument*) → int
Sets the monitor mode (bitmask integer - see DSS Help). Bitmask integer designating the values the monitor is to capture: 0 = Voltages and currents at designated terminal 1 = Powers at designated terminal 2 = Tap Position (Transformer Device only) 3 = State Variables (PCElements only) 4 = Flicker level and severity index (Pst) for voltages. No adders apply.

Flicker level at simulation time step, Pst at 10-minute time step.

5 = Solution variables (Iterations, etc). Normally, these would be actual phasor quantities from solution. 6 = Capacitor Switching (Capacitors only) 7 = Storage state vars (Storage device only) 8 = All winding currents (Transformer device only) 9 = Losses, watts and var (of monitored device) 10 = All Winding voltages (Transformer device only) Normally, these would be actual phasor quantities from solution. 11 = All terminal node voltages and line currents of monitored device Combine mode with adders below to achieve other results for terminal quantities: +16 = Sequence quantities +32 = Magnitude only +64 = Positive sequence only or avg of all phases

Mix adder to obtain desired results. For example: Mode=112 will save positive sequence voltage and current magnitudes only Mode=48 will save all sequence voltages and currents, but magnitude only.

monitors_write_terminal(*argument*) → int
Sets sets the terminal number of element being monitored.

class py_dss_interface.models.Monitors.Monitors.**MonitorsS**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: CStr MonitorsS(int32_t Parameter, CStr Argument);

This interface returns a string according to the number sent in the variable "parameter". The parameter can be one of the following.

monitors_file_name() → str
Returns the name of the CSV file associated with active monitor.

monitors_read_element() → str
Returns the full name of element being monitored by the active Monitor.

monitors_read_name() → str
Returns the active Monitor object by name.

monitors_write_element(*argument*) → str
Sets the full name of element being monitored by the active Monitor.

monitors_write_name(*argument*) → str
Sets the active Monitor object by name.

class py_dss_interface.models.Monitors.Monitors.**MonitorsV**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object. The structure of the interface is as follows:

void MonitorsV(int32_t Parameter, VARIANT *Argument);

This interface returns a variant according to the number sent in the variable "parameter". The parameter can be one of the following.

monitors_all_names()
Returns an array of all Monitor names (array of strings).

monitors_byte_stream()
Returns a byte array containing monitor stream values. Make sure a "save" is done first (standard solution modes do this automatically).

monitors_channel(*argument*) → str
Returns a variant array of doubles for the specified channel (usage: MyArray = DSSmonitor. Channel(i)) A save or SaveAll should be executed first. Done automatically by most standard solution modes.

monitors_dbl_freq()
Returns a variant array of doubles containing time values for harmonics mode solutions; empty for time mode solutions (use dblHour).

monitors_dbl_hour()
Returns returns a variant array of doubles containing time value in hours for the time-sampled monitor values; empty if frequency-sampled values for harmonics solution (see dblFreq).

monitors_header()
Returns the header string; Variant array of strings containing Channel Names.

4.1.24 Parallel

Created by eniocc at 11/10/2020

class py_dss_interface.models.Parallel.Parallel.**Parallel**(*obj_dss*)
Bases: py_dss_interface.models.Parallel.ParallelI.ParallelI, py_dss_interface.models.Parallel.ParallelV.ParallelV

These interfaces allows users to use the parallel processing features included in OpenDSS-PM. With this interface it is possible to create multiple actors, specify the CPU where the actor will be executed, control the execution of the actors, check the actors status and progress among many other functionalities.

class py_dss_interface.models.Parallel.Parallel.**ParallelI**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface allows to control parameters of the parallel computing suite of OpenDSS-PM where its value can be specified as an integer number.

The structure of the interface is as follows: int32_t ParallelI(int32_t Parameter, int32_t Argument);

This interface returns an integer number with the result of the query according to the value of the variable Parameter, which can be one of the following.

parallel_create_actor() → int
Creates a new actor and sets the active actor ID as the ID for the recently created actor. If there are no more CPUs available, the system will not allow the creation of the new actor.

parallel_num_actors() → int
Gets the number of actors created in the actual session.

parallel_num_cores() → int
Returns the number of physical cores available in the local computer. If your computer has less than 64 Cores, this number should be the number of CPUs/2. For more information, please check: https://www.howtogeek.com/194756/cpu-basics-multiple-cpus-cores-and-hyper-threading-explained/.

parallel_num_cpus() → int
Returns the number of CPUs available in the local computer.

parallel_read_active_actor() → int
Returns the ID of the active actor.

parallel_read_active_parallel() → int
Gets if the parallel features of OpenDSS-PM are active. If active, this parameter will return 1, otherwise, will return 0 and OpenDSS-PM will behave sequentially.

parallel_read_actor_cpu() → int
Gets the ID of the CPU assigned for the execution of the active actor.

parallel_read_concatenate_reportsl() → int
Gets the state of the ConcatenateReports property of OpenDSS-PM. If 1, means that every time the user executes a Show/Export monitor operation, the data stored on the monitors with the same name for each actor will be concatenated one after the other. Otherwise (0), to get access of each monitor the user will have to activate the actor of interest and then perform the Show/Export command on the desired monitor.

parallel_wait() → int
Waits until all the actors are free and ready to receive a new command.

parallel_write_active_actor(*argument*) → int
Sets the ID of the active actor; this number cannot be higher than the number of existing actors.

parallel_write_active_parallel(*argument*) → int
Sets enables/disables the parallel features of OpenDSS-PM. To enable set the argument in 1, otherwise, the argument should be 0 and OpenDSS-PM will behave sequentially.

parallel_write_actor_cpu(*argument*) → int
Sets the CPU for the execution of the active actor.

parallel_write_concatenate_reportsl(*argument*) → int
Sets the state of the ConcatenateReports property of OpenDSS-PM. If 1, means that every time the user executes a Show/Export monitor operation, the data stored on the monitors with the same name for each actor will be concatenated one after the other. Otherwise (0), to get access of each monitor the user will have to activate the actor of interest and then perform the Show/Export command on the desired monitor.

class py_dss_interface.models.Parallel.Parallel.**ParallelV**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: void ParallelV(int32_t Parameter, VARIANT *Argument);

This interface returns a Variant with the result of the query according to the value of the variable Parameter, which can be one of the following.

parallel_actor_progress()
Returns an array of integers containing the progress in percentage for each active actor.

parallel_actor_status()
Returns an array of integers containing the status of each active actor. If 1, the actor is ready to receive new commands, if 0, the actor is busy performing a simulation and cannot take new ?solve? commands at this time. However, the actor is capable to deliver values while the simulation is being performed.

4.1.25 Parser

Created by eniocc at 11/10/2020

class py_dss_interface.models.Parser.Parser.**Parser**(*obj_dss*)

Bases: py_dss_interface.models.Parser.ParserI.ParserI, py_dss_interface.models.Parser.ParserS.ParserS, py_dss_interface.models.Parser.ParserF.ParserF, py_dss_interface.models.Parser.ParserV.ParserV

This interface implements the CmathLib (ICmathLib) interface of OpenDSS by declaring 4 procedures for accessing the different properties included in this interface: ParserI, ParserS, ParserF, ParserV.

parser_matrix()
Use this property to parse a Matrix token in OpenDSS format. Returns square matrix of order specified. Order same as default fortran order: column by column.

parser_symmatrix()
Use this property to parse a Matrix token in lower triangular form. Symmetry is forced.

parser_vector()
Returns token as variant array of doubles. For parsing quoted array syntax.

class py_dss_interface.models.Parser.Parser.**ParserF**(*obj_dss*)

Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: double ParserF(int32_t Parameter, double Argument);

This interface returns a floating point number with the result of the query according to the value of the variable Parameter, which can be one of the following.

parser_dbl_value() → float
Returns next parameter as a double.

class py_dss_interface.models.Parser.Parser.**ParserI**(*obj_dss*)

Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: int32_t ParserI(int32_t Parameter, int32_t Argument);

This interface returns an integer number with the result of the query according to the value of the variable Parameter, which can be one of the following.

parser_int_value() → int
Returns next parameter as a long integer.

parser_read_auto_increment() → int
In this parameter the default is false (0). If true (1) parser automatically advances to next token after DblValue, IntValue, or StrValue. Simpler when you don't need to check for parameter names.

parser_reset_delimeters() → int
Reset delimiters to their default values.

parser_write_auto_increment(*argument*) → int
In this parameter the default is false (0). If true (1) parser automatically advances to next token after DblValue, IntValue, or StrValue. Simpler when you don't need to check for parameter names.

class py_dss_interface.models.Parser.Parser.**ParserS**(*obj_dss*)

Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: CStr ParserS(int32_t Parameter, CStr Argument);

This interface returns a string with the result of the query according to the value of the variable Parameter, which can be one of the following.

parser_next_param()
Gets next token and return tag name (before = sign) if any. See Autoincrement.

parser_read_begin_quote()
Gets the string containing the characters for quoting in OpenDSS scripts. Matching pairs defined in EndQuote. Default is "([{.

parser_read_cmd_string()
Gets a string to be parsed. Loading this string resets the parser to the beginning of the line. Then parse off the tokens in sequence.

parser_read_delimiters()
Gets the string defining hard delimiters used to separate token on the command string. Default is , and =. The = separates token name from token value. These override whitespaces to separate tokens.

parser_read_end_quote()
Gets the string containing the characters, in order, that match the beginning quote characters in BeginQuote. Default is ")]}.

parser_read_white_space()
Gets the characters used for White space in the command string. Default in blank and Tab.

parser_str_value()
Returns next parameter as a string.

parser_write_begin_quote(*argument*)
Sets the string containing the characters for quoting in OpenDSS scripts. Matching pairs defined in EndQuote. Default is "([{.

parser_write_cmd_string(*argument*)
Sets a string to be parsed. Loading this string resets the parser to the beginning of the line. Then parse off the tokens in sequence.

parser_write_delimiters(*argument*)
Sets the string defining hard delimiters used to separate token on the command string. Default is , and =. The = separates token name from token value. These override whitespace to separate tokens.

parser_write_end_quote(*argument*)
Sets the string containing the characters, in order, that match the beginning quote characters in BeginQuote. Default is ")]}.

parser_write_white_space(*argument*)
Sets the characters used for White space in the command string. Default in blank and Tab.

class py_dss_interface.models.Parser.Parser.**ParserV**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: void ParserV(int32_t Parameter, VARIANT *Argument);

This interface returns a Variant with the result of the query according to the value of the variable Parameter, which can be one of the following.

parser_matrix()
Use this property to parse a Matrix token in OpenDSS format. Returns square matrix of order specified. Order same as default fortran order: column by column.

parser_sym_matrix()
Use this property to parse a Matrix token in lower triangular form. Symmetry is forced.

parser_vector()
Returns token as variant array of doubles. For parsing quoted array syntax.

4.1.26 PDElements

Created by eniocc at 11/10/2020

class py_dss_interface.models.PDElements.PDElements.**PDElements**(*obj_dss*)
Bases: py_dss_interface.models.PDElements.PDElementsF.PDElementsF, py_dss_interface.models.PDElements.PDElementsI.PDElementsI, py_dss_interface.models.PDElements.PDElementsS.PDElementsS

This interface implements the PDElements (IPDElements) interface of OpenDSS by declaring 3 procedures for accessing the different properties included in this interface: PDElementsF, PDElementsI, PDElementsS.

class py_dss_interface.models.PDElements.PDElements.**PDElementsF**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: double PDElementsF(int32_t Parameter, double Argument);

This interface returns a floating point number with the result of the query according to the value of the variable Parameter, which can be one of the following.

pdelements_accumulated_l() → float
Gets the accumulated failure rate for this branch on down line.

pdelements_lambda() → float
Gets the failure rate for this branch. Faults per year including length of line.

pdelements_read_fault_rate() → float
Gets the number of failures per year. For LINE elements: Number of failures per unit length per year.

pdelements_read_pct_permanent() → float
Gets the percent of faults that are permanent (require repair). Otherwise, fault is assumed to be transient/temporary.

pdelements_repair_time() → float
Gets the average time to repair a permanent fault on this branch, hours.

pdelements_total_miles() → float
Gets the total miles of line from this element to the end of the zone. For recloser siting algorithm.

pdelements_write_fault_rate(*argument*) → float
Sets the number of failures per year. For LINE elements: Number of failures per unit length per year.

pdelements_write_pct_permanent(*argument*) → float
Sets the percent of faults that are permanent (require repair). Otherwise, fault is assumed to be transient/temporary.

class py_dss_interface.models.PDElements.PDElements.**PDElementsI**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: int32_t PDElementsI(int32_t Parameter, int32_t Argument);

This interface returns an integer with the result of the query according to the value of the variable Parameter, which can be one of the following.

pdelements_count() → int
Gets number of PDElements in active circuit.

pdelements_first() → int
Sets the first enabled PD element to be the active element. Returns 0 if none found.

pdelements_from_terminal() → int
Gets the number of the terminal of active PD element that is on the "from" side. This is set after the meter zone is determined.

pdelements_is_shunt() → int
Sets returns 1 if the PD element should be treated as a shunt element rather than a series element. Applies to capacitor and reactor elements in particular.

pdelements_next() → int
Sets the next enabled PD element to be the active element. Returns 0 if none found.

pdelements_num_customers() → int
Gets the number of customers in this branch.

pdelements_parent_pd_element() → int
Gets the parent PD element to be the active circuit element. Returns 0 if no more elements upline.

pdelements_section_id() → int
Gets the integer ID of the feeder section that this PDElement branch is part of.

pdelements_total_customers() → int
Gets the total number of customers from this branch to the end of the zone.

class py_dss_interface.models.PDElements.PDElements.**PDElementsS**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: CStr PDElementsF(int32_t Parameter, CStr Argument);

This interface returns a string with the result of the query according to the value of the variable Parameter, which can be one of the following.

pdelements_read_name()
Gets the name of the active PDElement, returns null string if active element id not PDElement.

pdelements_write_name(*argument*)
Sets the name of the active PDElement, returns null string if active element id not PDElement.

4.1.27 PVSystems

Created by eniocc at 11/10/2020

class py_dss_interface.models.PVSystems.PVSystems.**PVSystems**(*obj_dss*)
Bases: py_dss_interface.models.PVSystems.PVSystemsV.PVSystemsV, py_dss_interface.models.PVSystems.PVSystemsS.PVSystemsS, py_dss_interface.models.PVSystems.PVSystemsI.PVSystemsI, py_dss_interface.models.PVSystems.PVSystemsF.PVSystemsF

This interface implements the PVSystems (IPVSystems) interface of OpenDSS by declaring 4 procedures for accessing the different properties included in this interface: PVSystemsV, PVSystemsS, PVSystemsI, PVSystemsF.

class py_dss_interface.models.PVSystems.PVSystems.**PVSystemsF**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: double PVSystemsF(int32_t Parameter, double Argument);

This interface returns a floating point number with the result of the query according to the value of the variable Parameter, which can be one of the following.

pvsystems_kw() → float
Gets the kW output.

pvsystems_read_irradiance() → float
Gets the present value of the Irradiance property in W/sq-m.

pvsystems_read_kva_rated() → float
Gets the rated kVA of the PVSystem.

pvsystems_read_kvar() → float
Gets the kvar value.

pvsystems_read_pf() → float
Gets the power factor value.

pvsystems_read_pmpp() → float
Gets the Pmpp.

pvsystems_write_irradiance(*argument*) → float
Sets the present value of the Irradiance property in W/sq-m.

pvsystems_write_kva_rated(*argument*) → float
Sets the rated kVA of the PVSystem.

pvsystems_write_kvar(*argument*) → float
Sets the kvar value.

pvsystems_write_pf(*argument*) → float
Sets the power factor value.

pvsystems_write_pmpp(*argument*) → float
Sets the Pmpp.

class py_dss_interface.models.PVSystems.PVSystems.**PVSystemsI**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: int32_t PVSystemsI(int32_t Parameter, int32_t Argument);

This interface returns an integer number with the result of the query according to the value of the variable Parameter, which can be one of the following.

pvsystems_count() → int
Returns the number of PVSystem objects currently defined in the active circuit.

pvsystems_first() → int
Sets the first PVSystem to be active; returns 0 if none.

pvsystems_next() → int
Sets the next PVSystem to be active; returns 0 if none.

pvsystems_read_idx() → int
Gets the active PVSystem by index; 1..Count.

pvsystems_write_idx(*argument*) → int
Sets the active PVSystem by index; 1..Count..

class py_dss_interface.models.PVSystems.PVSystems.**PVSystemsS**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: CStr PVSystemsS(int32_t Parameter, CStr Argument);

This interface returns a string with the result of the query according to the value of the variable Parameter, which can be one of the following.

pvsystems_read_name()
Gets the name of the active PVSystem.

pvsystems_write_name(*argument*)
Sets the name of the active PVSystem.

class py_dss_interface.models.PVSystems.PVSystems.**PVSystemsV**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: void PVSystemsV(int32_t Parameter, VARIANT *Argument);

This interface returns a Variant with the result of the query according to the value of the variable Parameter, which can be one of the following.

pvsystems_all_names()
Gets the variant array of string containing names of all PVSystems in the circuit.

4.1.28 Reclosers

class py_dss_interface.models.Reclosers.Reclosers.**Reclosers**(*obj_dss*)
Bases: py_dss_interface.models.Reclosers.ReclosersI.ReclosersI, py_dss_interface.models.Reclosers.ReclosersV.ReclosersV, py_dss_interface.models.Reclosers.ReclosersS.ReclosersS, py_dss_interface.models.Reclosers.ReclosersF.ReclosersF

This interface implements the Reclosers (IReclosers) interface of OpenDSS by declaring 4 procedures for accessing the different properties included in this interface: ReclosersI, ReclosersV, ReclosersS, ReclosersF.

class py_dss_interface.models.Reclosers.Reclosers.**ReclosersF**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: double ReclosersF(int32_t Parameter, double Argument);

This interface returns a floating point number with the result of the query according to the value of the variable Parameter, which can be one of the following.

reclosers_read_ground_inst() → float
Gets the ground (3I0) instantaneous trip setting - curve multiplier or actual amps.

reclosers_read_ground_trip() → float
Gets the ground (3I0) trip multiplier or actual amps.

reclosers_read_phase_inst() → float
Gets the phase instantaneous curve multiplier or actual amps.

reclosers_read_phase_trip() → float
Gets the phase trip curve multiplier or actual amps.

reclosers_write_ground_inst(*argument*) → float

reclosers_write_ground_trip(*argument*) → float
Sets the ground (3I0) trip multiplier or actual amps.

reclosers_write_phase_inst(*argument*) → float
Sets the phase instantaneous curve multiplier or actual amps.

reclosers_write_phase_trip(*argument*) → float
Sets the phase trip curve multiplier or actual amps.

class py_dss_interface.models.Reclosers.Reclosers.**RecloserI**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: int32_t RecloserI(int32_t Parameter, int32_t Argument);

This interface returns an integer with the result of the query according to the value of the variable Parameter, which can be one of the following.

reclosers_close() → int
Close the switched object controlled by the recloser. Resets recloser to first operation.

reclosers_count() → int
Gets number of Reclosers in active circuit.

reclosers_first() → int
Sets first recloser to be active Circuit Element. Returns 0 if none.

reclosers_next() → int
Sets next recloser to be active Circuit Element. Returns 0 if none.

reclosers_open() → int
Open recloser's controlled element and lock out the recloser.

reclosers_read_idx() → int
Gets the active recloser by index into the recloser list. 1..Count.

reclosers_read_monitored_term() → int
Gets the terminal number of Monitored Object for the Recloser.

reclosers_read_num_fast() → int
Gets the number of fast shots.

reclosers_read_shots() → int
Gets the number of shots to lockout (fast + delayed).

reclosers_read_switched_term() → int
Gets the terminal of the controlled device being switched by the Recloser.

reclosers_write_idx(*argument*) → int
Sets the active recloser by index into the recloser list. 1..Count.

reclosers_write_monitored_term(*argument*) → int
Sets the terminal number of Monitored Object for the Recloser.

reclosers_write_num_fast(*argument*) → int
Sets the number of fast shots.

reclosers_write_shots(*argument*) → int
Sets the number of shots to lockout (fast + delayed).

reclosers_write_switched_term(*argument*) → int
Sets the terminal of the controlled device being switched by the Recloser.

class py_dss_interface.models.Reclosers.Reclosers.**RecloserssS**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: CStr RecloserssS(int32_t Parameter, CStr Argument);

This interface returns a string with the result of the query according to the value of the variable Parameter, which can be one of the following.

reclosers_read_monitored_obj()
Gets the full name of object this Recloser is monitoring.

reclosers_read_name()
Gets the name of the active Recloser Object.

reclosers_read_switched_obj()
Gets the full name of the circuit element that is being switched by this Recloser.

reclosers_write_monitored_obj(*argument*)
Sets the full name of object this Recloser is monitoring.

reclosers_write_name(*argument*)
Sets the name of the active Recloser Object.

reclosers_write_switched_obj(*argument*)
Sets the full name of the circuit element that is being switched by this Recloser.

class py_dss_interface.models.Reclosers.Reclosers.**RecloserssV**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: void RecloserssV(int32_t Parameter, VARIANT *Argument);

This interface returns a Variant with the result of the query according to the value of the variable Parameter, which can be one of the following.

reclosers_all_names()
Gets a variant array of strings with names of all reclosers in active circuit.

reclosers_reclose_intervals()
Gets a variant array of doubles: reclose intervals (s) between shots.

4.1.29 RegControls

Created by eniocc at 11/10/2020

class py_dss_interface.models.RegControls.RegControls.**RegControls**(*obj_dss*)
Bases: py_dss_interface.models.RegControls.RegControlsI.RegControlsI, py_dss_interface.models.RegControls.RegControlsF.RegControlsF, py_dss_interface.models.RegControls.RegControlsV.RegControlsV, py_dss_interface.models.RegControls.RegControlsS.RegControlsS

This interface implements the RegControls (IRegControls) interface of OpenDSS by declaring 4 procedures for accessing the different properties included in this interface: RegControlsI, RegControlsF, RegControlsV, RegControlsS.

class py_dss_interface.models.RegControls.RegControls.**RegControlsF**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: double RegControlsF(int32_t Parameter, double Argument);

This interface returns a floating point number with the result of the query according to the value of the variable Parameter, which can be one of the following.

regcontrols_read_ct_primary() → float
Gets the CT primary ampere rating (secondary is 0.2 amperes).

regcontrols_read_delay() → float
Gets the time delay [s] after arming before the first tap change. Control may reset before actually changing taps.

regcontrols_read_forward_band() → float
Gets the regulation bandwidth in forward direction, centered on Vreg.

regcontrols_read_forward_r() → float
Gets the LDC R settings in Volts.

regcontrols_read_forward_vreg() → float
Gets the target voltage in the forward direction, on PT secondary base.

regcontrols_read_forward_x() → float
Gets the LDC X settings in Volts.

regcontrols_read_pt_ratio() → float
Gets the PT ratio for voltage control settings.

regcontrols_read_reverse_band() → float
Gets the bandwidth in reverse direction, centered on reverse Vreg.

regcontrols_read_reverse_r() → float
Gets the reverse LDC R settings in Volts.

regcontrols_read_reverse_vreg() → float
Gets the target voltage in the reverse direction, on PT secondary base.

regcontrols_read_reverse_x() → float
Gets the reverse LDC X settings in Volts.

regcontrols_read_tap_delay() → float
Gets the time delay [s] for subsequent tap changes in a set. Control may reset before actually changing taps.

regcontrols_read_voltage_limit() → float
Gets the first house voltage limit on PT secondary base. Setting to 0 disables this function.

regcontrols_write_ct_primary(*argument*) → float
Sets the CT primary ampere rating (secondary is 0.2 amperes).

regcontrols_write_delay(*argument*) → float
Sets the time delay [s] after arming before the first tap change. Control may reset before actually changing taps.

regcontrols_write_forward_band(*argument*) → float
Sets the regulation bandwidth in forward direction, centered on Vreg.

regcontrols_write_forward_r(*argument*) → float
Sets the LDC R settings in Volts.

regcontrols_write_forward_vreg(*argument*) → float
Sets the target voltage in the forward direction, on PT secondary base.

regcontrols_write_forward_x(*argument*) → float
Sets sets the LDC X settings in Volts.

regcontrols_write_pt_ratio(*argument*) → float
Sets the PT ratio for voltage control settings.

regcontrols_write_reverse_band(*argument*) → float
Sets the bandwidth in reverse direction, centered on reverse Vreg.

regcontrols_write_reverse_r(*argument*) → float
Sets the reverse LDC R settings in Volts.

regcontrols_write_reverse_vreg(*argument*) → float
Sets the target voltage in the reverse direction, on PT secondary base.

regcontrols_write_reverse_x(*argument*) → float
Sets the reverse LDC X settings in Volts.

regcontrols_write_tap_delay(*argument*) → float
Sets the time delay [s] for subsequent tap changes in a set. Control may reset before actually changing taps.

regcontrols_write_voltage_limit(*argument*) → float
Sets the first house voltage limit on PT secondary base. Setting to 0 disables this function.

class py_dss_interface.models.RegControls.RegControls.**RegControlsI**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: int32_t RegControlsI(int32_t Parameter, int32_t Argument);

This interface returns an integer with the result of the query according to the value of the variable Parameter, which can be one of the following.

regcontrols_count() → int
Gets the number of RegControl objects in Active Circuit.

regcontrols_first() → int
Sets the first RegControl active. Returns 0 if no more.

regcontrols_next() → int
Sets the next RegControl active. Returns 0 if no more

regcontrols_read_is_inverse_time() → int
Gets the inverse time feature. Time delay is inversely adjusted, proportional to the amount of voltage outside the regulator band.

regcontrols_read_is_reversible() → int
Gets the setting in the reverse direction, usually not applicable to substation transformers.

regcontrols_read_max_tap_change() → int
Gets the maximum tap change per iteration in STATIC solution mode. 1 is more realistic, 16 is the default for faster solution.

regcontrols_read_tap_number() → int
Gets the actual tap number of the active RegControl.

regcontrols_read_tap_winding() → int
Gets the tapped winding number.

regcontrols_read_winding() → int
Gets the winding number for PT and CT connections.

regcontrols_write_is_inverse_time(*argument*) → int
Sets the inverse time feature. Time delay is inversely adjusted, proportional to the amount of voltage outside the regulator band.

regcontrols_write_is_reversible(*argument*) → int
Sets the different settings for the reverse direction (see Manual for details), usually not applicable to substation transformers.

regcontrols_write_max_tap_change(*argument*) → int
Sets the maximum tap change per iteration in STATIC solution mode. 1 is more realistic, 16 is the default for faster solution.

regcontrols_write_tap_number(*argument*) → int
Sets the actual tap number of the active RegControl.

regcontrols_write_tap_winding(*argument*) → int
Sets the tapped winding number.

regcontrols_write_winding(*argument*) → int
Sets the winding number for PT and CT connections.

class py_dss_interface.models.RegControls.RegControls.**RegControlsS**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: CStr RegControlsS(int32_t Parameter, CStr Argument);

This interface returns a string with the result of the query according to the value of the variable Parameter, which can be one of the following.

regcontrols_read_monitored_bus()
Gets the name of the remote regulated bus, in lieu of LDC settings.

regcontrols_read_name()
Gets the active RegControl name.

regcontrols_read_transformer()
Gets the name of the transformer this regulator controls.

regcontrols_write_monitored_bus(*argument*)
Sets the name of the remote regulated bus, in lieu of LDC settings.

regcontrols_write_name(*argument*)
Sets the active RegControl name.

regcontrols_write_transformer(*argument*)
Sets the name of the transformer this regulator controls.

class py_dss_interface.models.RegControls.RegControls.**RegControlsV**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: void RegControlsV(int32_t Parameter, VARIANT *Argument);

This interface returns a Variant with the result of the query according to the value of the variable Parameter, which can be one of the following.

regcontrols_all_names()
Gets a variant array of strings containing all RegControl names.

4.1.30 Relays

Created by eniocc at 11/10/2020

class py_dss_interface.models.Relays.Relays.**Relays**(*obj_dss*)
Bases: py_dss_interface.models.Relays.RelaysS.RelaysS, py_dss_interface.models.Relays.RelaysV.RelaysV, py_dss_interface.models.Relays.RelaysI.RelaysI

This interface implements the Relays (IRelays) interface of OpenDSS by declaring 3 procedures for accessing the different properties included in this interface: RelaysS, RelaysV, RelaysI.

class py_dss_interface.models.Relays.Relays.**RelaysI**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: int32_t RelaysI(int32_t Parameter, int32_t Argument);

This interface returns an integer with the result of the query according to the value of the variable Parameter, which can be one of the following.

relays_count() → int
Gets number of Relays in active circuit.

relays_first() → int
Sets first relay active. If none, returns 0.

relays_next() → int
Sets next relay active. If none, returns 0.

relays_read_idx() → int
Gets the active relay by index into the Relay list. 1..Count.

relays_read_monitored_term() → int
Gets the number of terminal of monitored element that this relay is monitoring.

relays_read_switched_term() → int
Gets the number of terminal of the switched object that will be opened when the relay trips.

relays_write_idx(*argument*) → int
Sets the active relay by index into the Relay list. 1..Count.

relays_write_monitored_term(*argument*) → int
Sets the number of terminal of monitored element that this relay is monitoring.

relays_write_switched_term(*argument*) → int
Sets the number of terminal of the switched object that will be opened when the relay trips.

class py_dss_interface.models.Relays.Relays.**RelaysS**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: CStr RelaysS(int32_t Parameter, CStr Argument);

This interface returns a string with the result of the query according to the value of the variable Parameter, which can be one of the following.

relays_read_monitored_obj()
Gets the full name of the object this relay is monitoring.

relays_read_name()
Gets the name of the active Relay.

relays_read_switched_obj()
Gets the full name of element that will switched when relay trips.

relays_write_monitored_obj(*argument*)
Sets the full name of the object this relay is monitoring.

relays_write_name(*argument*)
Sets the name of the active Relay.

relays_write_switched_obj(*argument*)
Sets the full name of element that will switched when relay trips.

class py_dss_interface.models.Relays.Relays.**RelaysV**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: void RelaysV(int32_t Parameter, VARIANT *Argument);

This interface returns a Variant with the result of the query according to the value of the variable Parameter, which can be one of the following.

relays_all_names()
Getsa variant array of strings containing names of all relay elements.

4.1.31 Sensors

Created by eniocc at 11/10/2020

class py_dss_interface.models.Sensors.Sensors.**Sensors**(*obj_dss*)
Bases: py_dss_interface.models.Sensors.SensorsV.SensorsV, py_dss_interface.models.Sensors.SensorsS.SensorsS, py_dss_interface.models.Sensors.SensorsI.SensorsI, py_dss_interface.models.Sensors.SensorsF.SensorsF

This interface implements the Sensors (ISensors) interface of OpenDSS by declaring 4 procedures for accessing the different properties included in this interface: SensorsV, SensorsS, SensorsI, SensorsF.

class py_dss_interface.models.Sensors.Sensors.**SensorsF**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: double SensorsF(int32_t Parameter, double Argument);

This interface returns a floating point number with the result of the query according to the value of the variable Parameter, which can be one of the following.

sensors_read_kv_base() → float
Gets the voltage base for the sensor measurements. LL for 2 and 3 - phase sensors, LN for 1-phase sensors.

sensors_read_pct_error() → float
Gets the assumed percent error in the Sensor measurement. Default is 1.

sensors_read_weight() → float
Gets the weighting factor for this sensor measurement with respect to the other sensors. Default is 1.

sensors_write_kv_base(*argument*) → float
Sets the voltage base for the sensor measurements. LL for 2 and 3 - phase sensors, LN for 1-phase sensors.

sensors_write_pct_error(*argument*) → float
Sets the assumed percent error in the Sensor measurement. Default is 1.

sensors_write_weight(*argument*) → float
Sets the weighting factor for this sensor measurement with respect to the other sensors. Default is 1.

class py_dss_interface.models.Sensors.Sensors.**SensorsI**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: int32_t SensorsI(int32_t Parameter, int32_t Argument);

This interface returns an integer with the result of the query according to the value of the variable Parameter, which can be one of the following.

sensors_count() → int
Gets number of sensors in active circuit.

sensors_first() → int
Sets the first sensor active. Returns 0 if none.

sensors_next() → int
Sets the next sensor active. Returns 0 if none.

sensors_read_is_delta() → int
Returns 1 if the sensor is connected in delta; otherwise, returns 0.

sensors_read_metered_terminal() → int
Gets the number of the measured terminal in the measured element.

sensors_read_reverse_delta() → int
Returns 1 if voltage measurements are 1-3, 3-2, 2-1; otherwise 0.

sensors_reset() → int
Clears the active sensor.

sensors_reset_all() → int
Clears all sensors in the active circuit.

sensors_write_is_delta(*argument*) → int
Allows to set 1 if the sensor is connected in delta; otherwise, set 0 (argument).

sensors_write_metered_terminal(*argument*) → int
Sets the number of the measured terminal in the measured element.

sensors_write_reverse_delta(*argument*) → int
Allows to set 1 if voltage measurements are 1-3, 3-2, 2-1; otherwise 0.

class py_dss_interface.models.Sensors.Sensors.**SensorsS**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: CStr SensorsS(int32_t Parameter, CStr Argument); This interface returns a string with the result of the query according to the value of the variable Parameter, which can be one of the following.

sensors_read_metered_element()
Gets the full name of the measured element.

sensors_read_name()
Gets the name of the active sensor object.

sensors_write_metered_element(*argument*)
Sets the full name of the measured element.

sensors_write_name(*argument*)
Sets the name of the active sensor object.

class py_dss_interface.models.Sensors.Sensors.**SensorsV**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: void SensorsV(int32_t Parameter, VARIANT *Argument);

This interface returns a Variant with the result of the query according to the value of the variable Parameter, which can be one of the following.

sensors_all_names()
Returns a variant array of sensor names.

sensors_read_currents()
Gets an array of doubles for the line current measurements; don't use with KWS and KVARS.

sensors_read_kvars()
Gets an array of doubles for Q measurements; overwrites currents with a new estimate using KWS.

sensors_read_kws()
Gets an array of doubles for P measurements; overwrites currents with a new estimate using KVARS.

sensors_write_currents(*argument*)
Sets an array of doubles for the line current measurements; don't use with KWS and KVARS.

sensors_write_kvars(*argument*)
Sets an array of doubles for Q measurements; overwrites currents with a new estimate using KWS.

sensors_write_kws(*argument*)
Sets an array of doubles for P measurements; overwrites currents with a new estimate using KVARS.

4.1.32 Settings

Created by eniocc at 11/10/2020

class py_dss_interface.models.Settings.Settings.**Settings**(*obj_dss*)
Bases: py_dss_interface.models.Settings.SettingsS.SettingsS, py_dss_interface.models.Settings.SettingsF.SettingsF, py_dss_interface.models.Settings.SettingsI.SettingsI, py_dss_interface.models.Settings.SettingsV.SettingsV

This interface implements the Settings (ISettings) interface of OpenDSS by declaring 4 procedures for accessing the different properties included in this interface: SettingsS, SettingsF, SettingsI, SettingsV.

class py_dss_interface.models.Settings.Settings.**SettingsF**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: double SettingsF(int32_t Parameter, double Argument);

This interface returns a floating point number with the result of the query according to the value of the variable Parameter, which can be one of the following.

settings_allocation_factors() → float
Sets all load allocation factors for all loads defined by XFKVA property to this value.

settings_read_emerg_vmax_pu() → float
Gets the per unit maximum voltage for Emergency conditions.

settings_read_emerg_vmin_pu() → float
Gets the per unit minimum voltage for Emergency conditions.

settings_read_loss_weight() → float
Gets the weighting factor applied to Loss register values.

settings_read_norm_vmax_pu() → float
Gets the per unit maximum voltage for Normal conditions.

settings_read_norm_vmin_pu() → float
Gets the per unit minimum voltage for Normal conditions.

settings_read_price_signal() → float
Gets the price signal for the circuit.

settings_read_ue_weight() → float
Gets the weighting factor applied to UE register values.

settings_write_emerg_vmax_pu(*argument*) → float
Sets the per unit maximum voltage for Emergency conditions.

settings_write_emerg_vmin_pu(*argument*) → float
Sets the per unit minimum voltage for Emergency conditions.

settings_write_loss_weight(*argument*) → float
Sets the weighting factor applied to Loss register values.

settings_write_norm_vmax_pu(*argument*) → float
Sets the per unit maximum voltage for Normal conditions.

settings_write_norm_vmin_pu(*argument*) → float
Sets the per unit minimum voltage for Normal conditions.

settings_write_price_signal(*argument*) → float
Sets the price signal for the circuit.

settings_write_ue_weight(*argument*) → float
Sets the weighting factor applied to UE register values.

class py_dss_interface.models.Settings.Settings.**SettingsI**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: int32_t SettingsI(int32_t Parameter, int32_t Argument);

This interface returns an integer with the result of the query according to the value of the variable Parameter, which can be one of the following.

settings_read_allow_duplicates() → int
Gets if OpenDSS allows duplicate names of objects: {1 allow, 0 not allow}.

settings_read_ckt_model() → int
Gets {dssMultiphase* | dssPositiveSeq} Indicate if the circuit model is positive sequence.

settings_read_trapezoidal() → int
Gets {True (1) | False (0)} value of trapezoidal integration flag in Energy Meters.

settings_read_zone_lock() → int
Gets the status of Lock zones on energy meters to prevent rebuilding if a circuit change occurs: {1= true, 0= False}.

settings_write_allow_duplicates(*argument*) → int
Sets if OpenDSS allows duplicate names of objects: {1 allow, 0 not allow}.

settings_write_ckt_model(*argument*) → int
Sets {dssMultiphase* | dssPositiveSeq} Indicate if the circuit model is positive sequence.

settings_write_trapezoidal(*argument*) → int
Sets {True (1) | False (0)} value of trapezoidal integration flag in Energy Meters.

settings_write_zone_lock(*argument*) → int
Sets the status of Lock zones on energy meters to prevent rebuilding if a circuit change occurs: {1= true, 0= False}.

class py_dss_interface.models.Settings.Settings.**SettingsS**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: CStr SettingsS(int32_t Parameter, CStr Argument);

This interface returns a string with the result of the query according to the value of the variable Parameter, which can be one of the following.

settings_read_auto_bus_list()
Gets the list of Buses or (File=xxxxx) syntax for the AutoAdd solution mode.

settings_read_price_curve()
Gets the name of LoadShape object that serves as the source of price signal data for yearly simulations, etc.

settings_write_auto_bus_list(*argument*)
Sets the list of Buses or (File=xxxxx) syntax for the AutoAdd solution mode.

settings_write_price_curve(*argument*)
Sets the name of LoadShape object that serves as the source of price signal data for yearly simulations, etc.

class py_dss_interface.models.Settings.Settings.**SettingsV**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: void SettingsV(int32_t Parameter, VARIANT *Argument);

This interface returns a Variant with the result of the query according to the value of the variable Parameter, which can be one of the following.

settings_read_loss_regs()
Gets the array of Integers defining Energy Meter registers to use for computing Losses.

settings_read_ue_regs()
Gets the array of Integers defining Energy Meter registers to use for computing UE.

settings_read_voltage_bases()
Gets the array of doubles defining the legal voltage bases in kV L-L.

settings_write_loss_regs(*argument*)
Sets the array of Integers defining Energy Meter registers to use for computing Losses.

settings_write_ue_regs(*argument*)
Sets the array of Integers defining Energy Meter registers to use for computing UE.

settings_write_voltage_bases(*argument*)
Sets the array of doubles defining the legal voltage bases in kV L-L.

4.1.33 Solution

Created by eniocc at 11/10/2020

class py_dss_interface.models.Solution.Solution.**Solution**(*obj_dss*)
Bases: py_dss_interface.models.Solution.SolutionI.SolutionI, py_dss_interface.models.Solution.SolutionF.SolutionF, py_dss_interface.models.Solution.SolutionS.SolutionS, py_dss_interface.models.Solution.SolutionV.SolutionV

This interface implements the Solution (ISolution) interface of OpenDSS by declaring 4 procedures for accessing the different properties included in this interface: SolutionI, SolutionF, SolutionS, SolutionV.

class py_dss_interface.models.Solution.Solution.**SolutionF**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: double SolutionF(int32_t Parameter, double Argument);

This interface returns a floating point number according to the number sent in the variable "parameter". The parameter can be one of the following.

solution_process_time() → float
Retrieves the time required (microseconds) to perform the latest solution time step, this time does not include the time required for sampling meters/monitors.

solution_process_time_step() → float
Retrieves the time required (microseconds) to perform the latest solution time step including the time required for sampling meters/monitors.

solution_read_cap_kvar() → float
Returns the capacitor kvar for adding in AutoAdd mode.

solution_read_dbl_hour() → float
Returns the hour as a double, including fractional part.

solution_read_frequency() → float
Returns the frequency for the next solution.

solution_read_gen_kw() → float
Returns the generator kW for AutoAdd mode.

solution_read_gen_mult() → float
Returns the default multiplier applied to generators (like LoadMult).

solution_read_gen_pf() → float
Returns the pf for generators in AutoAdd mode.

solution_read_load_mult() → float
Returns the default load multiplier applied to all non-fixed loads.

solution_read_pct_growth() → float
Returns the percent default annual load growth rate.

solution_read_seconds() → float
Returns the seconds from top of the hour.

solution_read_step_size() → float
Returns the step size for the next solution.

solution_read_tolerance() → float
Returns the solution convergence tolerance.

solution_read_total_time() → float
Retrieves the accumulated time required (microseconds) to perform the simulation.

solution_step_size_hr() → float
Sets the step size in Hours.

solution_step_size_min() → float
Sets the step size in minutes.

solution_write_cap_kvar(*argument*) → float
Sets the capacitor kvar for adding in AutoAdd mode.

solution_write_dbl_hour(*argument*) → float
Sets the hour as a double, including fractional part.

solution_write_frequency(*argument*) → float
Sets the frequency for the next solution.

solution_write_gen_kw(*argument*) → float
Sets the generator kW for AutoAdd mode.

solution_write_gen_mult(*argument*) → float
Sets the default multiplier applied to generators (like LoadMult).

solution_write_gen_pf(*argument*) → float
Sets the pf for generators in AutoAdd mode.

solution_write_load_mult(*argument*) → float
Sets the default load multiplier applied to all non-fixed loads.

solution_write_pct_growth(*argument*) → float
Sets the percent default annual load growth rate.

solution_write_seconds(*argument*) → float
Sets the seconds from top of the hour.

solution_write_step_size(*argument*) → float
Sets the step size for the next solution.

solution_write_tolerance(*argument*) → float
Sets the solution convergence tolerance.

solution_write_total_time(*argument*) → float
Sets the accumulated time (microseconds) register. The new value for this register must be specified in the argument.

class py_dss_interface.models.Solution.Solution.**SolutionI**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: int32_t SolutionI(int32_t Parameter, int32_t Argument);

This interface returns an integer according to the number sent in the variable "parameter". The parameter can be one of the following.

solution_build_y_matrix() → int
Forces building of the System Y matrix according to the argument: {1= series elements only | 2= Whole Y matrix}.

solution_calc_inc_matrix() → int
Starts the calculation of the incidence matrix for the active actor. Please be sure that the circuits of each actor have been compiled and ready to be solved before using this command.

solution_calc_inc_matrix_0() → int
Starts the calculation of the Branch to Node incidence matrix for the active actor. Please be sure that the circuits of each actor have been compiled and ready to be solved before using this command. The

difference between this command and the CalcIncMatrix is that the calculated matrix will be ordered hierarchically from the substation to the feeder end, which can be helpful for many operations. Additionally, the Bus Levels vector is calculated and the rows (PDElements) and columns (Buses) are permuted so it is easy to identify their position in the circuit.

solution_check_controls() → int
Performs the normal process for sampling and executing Control Actions and Fault Status and rebuilds Y if necessary.

solution_check_fault_status() → int
Executes status check on all fault objects defined in the circuit. Returns 0.

solution_clean_up() → int
Update storage, invcontrol, etc., at end of time step.

solution_do_control_actions() → int
Pops control actions off the control queue and dispatches to the proper control element.

solution_finish_time_step() → int
Calls cleanup, sample monitors, and increment time at end of time step.

solution_init_snap() → int
Initializes some variables for snap shot power flow. SolveSnap does this automatically.

solution_iterations() → int
Returns the number of iterations taken for the last solution.

solution_most_iterations_done() → int
Returns the max number of iterations required to converge at any control iteration of the most recent solution..

solution_read_add_type() → int
Returns the type of device to add in AutoAdd Mode: {dssGen (default)|dssCap}.

solution_read_algorithm() → int
Returns the base solution algorithm: {dssNormalSolve | dssNewtonSolve}.

solution_read_control_actions_done() → int
Indicates that the control actions are done: {1 done, 0 not done}.

solution_read_control_iterations() → int
Returns the current value of the control iteration counter.

solution_read_control_mode() → int
Returns the mode for control devices: {dssStatic (default) | dssEvent | dssTime}.

solution_read_converged() → int
Indicates whether the circuit solution converged (1 converged | 0 not converged).

solution_read_hour() → int
Returns the present hour (See DSS help).

solution_read_load_model() → int
Returns the Load Model: {dssPowerFlow (default)|dssAdmittance}.

solution_read_max_control_iterations() → int
Returns the maximum allowable control iterations.

solution_read_max_iterations() → int
Returns the Maximum number of iterations used to solve the circuit.

solution_read_mode() → int
Returns the present solution mode (See DSS help).

solution_read_number() → int
Returns the number of solutions to perform for MonteCarlo and time series simulations.

solution_read_random() → int
Returns the randomization mode for random variables "Gaussian" o "Uniform".

solution_read_year() → int
Returns the present Year (See DSS help).

solution_sample_control_devices() → int
Executes a sampling of all intrinsic control devices, which push control actions into the control queue.

solution_sample_do_control_actions() → int
Sample controls and then process the control queue for present control mode and dispatch control actions. Returns 0.

solution_solve() → int
Solution for the present solution mode. Returns 0.

solution_solve_all() → int
Starts the solution process for all the actors created in memory. Please be sure that the circuits of each actor have been compiled and ready to be solved before using this command.

solution_solve_direct() → int
Executes a direct solution from the system Y matrix, ignoring compensation currents of loads, generators (includes Yprim only).

solution_solve_no_control() → int
Is similar to SolveSnap except no control actions are checked or executed.

solution_solve_plus_control() → int
Executes a power flow solution (SolveNoControl) plus executes a CheckControlActions that executes any pending control actions.

solution_solve_power_flow() → int
Solves using present power flow method. Iterative solution rather than direct solution.

solution_system_y_changed() → int
Indicates if elements of the System Y have been changed by recent activity. If changed returns 1; otherwise 0.

solution_total_iterations() → int
Returns the total iterations including control iterations for most recent solution.

solution_write_add_type(*argument*) → int
Modifies the type of device to add in AutoAdd Mode: {dssGen (default)|dssCap}.

solution_write_algorithm(*argument*) → int
Modifies the base solution algorithm: {dssNormalSolve | dssNewtonSolve}.

solution_write_control_actions_done(*argument*) → int
Modifies the flag to indicate that the control actions are done: {1 done, 0 not done}.

solution_write_control_iterations(*argument*) → int
Modifies the current value of the control iteration counter.

solution_write_control_mode(*argument*) → int
Modifies the mode for control devices: {dssStatic (default) | dssEvent | dssTime}.

solution_write_converged(*argument*) → int
Modifies the converged flag (1 converged | 0 not converged).

solution_write_hour(*argument*) → int
Modifies the present hour (See DSS help).

solution_write_load_model(*argument*) → int
Modifies the Load Model: {dssPowerFlow (default)|dssAdmittance}.

solution_write_max_control_iterations(*argument*) → int
Modifies the maximum allowable control iterations.

solution_write_max_iterations(*argument*) → int
Modifies the Maximum number of iterations used to solve the circuit.

solution_write_mode(*argument*) → int
Modifies the present solution mode (See DSS help).

solution_write_number(*argument*) → int
Modifies the number of solutions to perform for MonteCarlo and time series simulations.

solution_write_random(*argument*) → int
Modifies the randomization mode for random variables "Gaussian" o "Uniform".

solution_write_year(*argument*) → int
Modifies the present Year (See DSS help).

class py_dss_interface.models.Solution.Solution.**SolutionS**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: CStr SolutionS(int32_t Parameter, CStr Argument);

This interface returns a string according to the number sent in the variable "parameter". The parameter can be one of the following.

solution_mode_id()
Returns the ID (text) of the present solution mode.

solution_read_default_daily()
Returns the default daily load shape (defaults to "Default").

solution_read_default_yearly()
Returns the default yearly load shape (defaults to "Default").

solution_read_ld_curve()
Returns the Load-Duration Curve name for LD modes.

solution_write_default_daily(*argument*)
Sets the default daily load shape (defaults to "Default").

solution_write_default_yearly(*argument*)
Sets the default yearly load shape (defaults to "Default").

solution_write_ld_curve(*argument*)
Sets the Load-Duration Curve name for LD modes.

class py_dss_interface.models.Solution.Solution.**SolutionV**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: void SolutionV(int32_t Parameter, VARIANT *Argument);

This interface returns a variant according to the number sent in the variable "parameter". The parameter can be one of the following.

solution_bus_levels()
Returns an array of integers containing BusLevels array. This array gives a numeric value to each bus to specify how far it is from the circuit?s backbone (a continuous path from the feeder head to the feeder end). It is very handy to understand the circuit?s topology.

solution_event_log()
Returns an array of strings containing the Event Log.

solution_inc_matrix_cols()
Returns an array of strings specifying the way the cols of the incidence matrix (buses) are organized, depending on the way the Branch to node incidence matrix was calculated (CalcIncMatrix/CalcIncMatrix_O) the result could be very different.

solution_inc_matrix_rows()
Returns an array of strings specifying the way the rows of the incidence matrix (PDElements) are organized, depending on the way the Branch to node incidence matrix was calculated (CalcIncMatrix/CalcIncMatrix_O) the result could be very different..

solution_laplacian()
Returns an array of integers containing the Laplacian matrix using the incidence matrix previously calculated , this means that before calling this command the incidence matrix needs to be calculated using calcincmatrix/calcincmatrix_o. This command will return only the non-zero values in compressed coordinate format (row, col, value)..

solution_nc_matrix()
Returns an array of integers containing the incidence matrix (1-D). Each cell of the incidence matrix is delivered using 3 elements of the array delivered, the first is the row, the second is the column and the third is the value (1/-1). This procedure will only deliver the non-zero elements..

4.1.34 SwtControls

Created by eniocc at 11/10/2020

class py_dss_interface.models.SwtControls.SwtControls.**SwtControls**(*obj_dss*)
Bases: py_dss_interface.models.SwtControls.SwtControlsS.SwtControlsS, py_dss_interface.models.SwtControls.SwtControlsV.SwtControlsV, py_dss_interface.models.SwtControls.SwtControlsI.SwtControlsI, py_dss_interface.models.SwtControls.SwtControlsF.SwtControlsF

This interface implements the SwtControls (ISwtControls) interface of OpenDSS by declaring 4 procedures for accessing the different properties included in this interface: SwtControlsS, SwtControlsV, SwtControlsI, SwtControlsF.

class py_dss_interface.models.SwtControls.SwtControls.**SwtControlsF**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: double SwtControlsF(int32_t Parameter, double Argument);

This interface returns a floating point number (64 bits) with the result of the query according to the value of the variable Parameter, which can be one of the following.

swtcontrols_read_delay() → float
Gets the time delay [s] between arming and opening or closing the switch. Control may reset before actually operating the switch.

swtcontrols_write_delay(*argument*) → float
Sets sets the time delay [s] between arming and opening or closing the switch. Control may reset before actually operating the switch.

class py_dss_interface.models.SwtControls.SwtControls.**SwtControlsI**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: int32_t SwtControlsI(int32_t Parameter, int32_t Argument);

This interface returns an integer with the result of the query according to the value of the variable Parameter, which can be one of the following.

swtcontrols_count() → int
Gets the total number of SwtControls in the active circuit.

swtcontrols_first() → int
Sets the first SwtControl active. Returns 0 if no more.

swtcontrols_next() → int
Sets the next SwtControl active. Returns 0 if no more.

swtcontrols_read_action() → int
Gets the open (1) or close (2) action of the switch. No effect if switch is locked. However, reset removes any lock and then closes the switch (shelf state). 0 = none action.

swtcontrols_read_is_locked() → int
Gets the lock state: {1 locked | 0 not locked}.

swtcontrols_read_switched_term() → int
Gets the terminal number where the switch is located on the SwitchedObj.

swtcontrols_write_action(*argument*) → int
Sets open (1) or close (2) the switch. No effect if switch is locked. However, reset removes any lock and then closes the switch (shelf state). 0 = none action (see manual for details).

swtcontrols_write_is_locked(*argument*) → int
Sets the lock to prevent both manual and automatic switch operation.

swtcontrols_write_switched_term(*argument*) → int
Sets the terminal number where the switch is located on the SwitchedObj.

class py_dss_interface.models.SwtControls.SwtControls.**SwtControlsS**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: CStr SwtControlsS(int32_t Parameter, CStr Argument);

This interface returns a string with the result of the query according to the value of the variable Parameter, which can be one of the following.

swtcontrols_read_name()
Gets the active swtcontrol name.

swtcontrols_read_switched_obj()
Gets the name of the switched object by the active SwtControl

swtcontrols_write_name(*argument*)
Sets the active swtcontrol by name.

swtcontrols_write_switched_obj(*argument*)
Sets the switched object by name.

class py_dss_interface.models.SwtControls.SwtControls.**SwtControlsV**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: void SwtControlsV(int32_t Parameter, VARIANT *Argument);

This interface returns a Variant with the result of the query according to the value of the variable Parameter, which can be one of the following.

swtcontrols_all_names()
Gets a variant array of strings with all SwtControl names in the active circuit.

4.1.35 Text

Created by eniocc at 11/10/2020

class py_dss_interface.models.Text.Text.**Text**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

text(*argument*)
Can be used to send commands to the text interface of OpenDSS (DSS.Text).

4.1.36 Topology

Created by eniocc at 11/10/2020

class py_dss_interface.models.Topology.Topology.**Topology**(*obj_dss*)
Bases: py_dss_interface.models.Topology.TopologyI.TopologyI, py_dss_interface.models.Topology.TopologyV.TopologyV, py_dss_interface.models.Topology.TopologyS.TopologyS

This interface implements the Topology (ITopology) interface of OpenDSS by declaring 3 procedures for accessing the different properties included in this interface: TopologyI, TopologyV, TopologyS.

class py_dss_interface.models.Topology.Topology.**TopologyI**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: int32_t TopologyI(int32_t Parameter, int32_t Argument);

This interface returns an integer with the result of the query according to the value of the variable Parameter, which can be one of the following.

topology_active_branch() → int
Returns the index of the active Branch.

topology_active_level() → int
Gets the topological depth of the active branch.

topology_backward_branch() → int
Moves back toward the source, return index of new active branch or 0 if no more.

topology_first() → int
Sets the first branch active, returns 0 if none.

topology_first_load() → int
Sets as active load the first load at the active branch, return index or 0 if none.

topology_forward_branch() → int
Moves forward in the tree, return index of new active branch or 0 if no more.

topology_looped_branch() → int
Moves to looped branch, return index or 0 if none.

topology_next() → int
Sets the next branch active, returns 0 if none.

topology_next_load() → int
Sets as active load the next load at the active branch, return index or 0 if none.

topology_num_isolated_branches() → int
Gets the number of isolated branches (PD elements and capacitors).

topology_num_isolated_loadss() → int
Gets the number of isolated loads.

topology_num_loops() → int
Gets the number of loops.

topology_parallel_branch() → int
Mode to directly parallel branch, return index or 0 if none.

class py_dss_interface.models.Topology.Topology.**TopologyS**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: CStr TopologyS(int32_t Parameter, CStr Argument);

This interface returns a string with the result of the query according to the value of the variable Parameter, which can be one of the following.

topology_read_branch_name()
Gets the name of the active branch.

topology_read_bus_name()
Gets the name of the active Bus.

topology_write_branch_name(*argument*)
Sets the name of the active branch.

topology_write_bus_name(*argument*)
Sets the Bus active by name.

class py_dss_interface.models.Topology.Topology.**TopologyV**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: void TopologyV(int32_t Parameter, VARIANT *Argument);

This interface returns a Variant with the result of the query according to the value of the variable Parameter, which can be one of the following.

topology_all_isolated_branches()
Gets a variant array of all isolated branch names.

topology_all_isolated_loads()
Gets a variant array of all isolated load names.

topology_all_looped_pairs()
Gets a variant array of all looped element names, by pairs.

4.1.37 Transformers

Created by eniocc at 11/10/2020

class py_dss_interface.models.Transformers.Transformers.**Transformers**(*obj_dss*)
Bases: py_dss_interface.models.Transformers.TransformersV.TransformersV, py_dss_interface.models.Transformers.TransformersF.TransformersF, py_dss_interface.models.Transformers.TransformersI.TransformersI, py_dss_interface.models.Transformers.TransformersS.TransformersS

This interface implements the Transformers (ITransformer) interface of OpenDSS by declaring 4 procedures for accessing the different properties included in this interface: TransformersV, TransformersF, TransformersI, TransformersS.

class py_dss_interface.models.Transformers.Transformers.**TransformersF**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/modify the properties of the Transformers Class where the values are doubles.

The structure of the interface is as follows: double TransformersF(int32_t Parameter, double argument) ;

This interface returns a floating point number (64 bits), the variable "parameter" is used to specify the property of the class to be used and the variable "argument" can be used to modify the value of the property when necessary. Reading and writing properties are separated and require a different parameter number to be executed.

The properties (parameter) are integer numbers and are described as follows.

transformers_read_kv() → float
Gets the active winding kV rating. Phase-phase for 2 or 3 phases, actual winding kV 1 phase transformer.

transformers_read_kva() → float
Gets the active winding kVA rating. On winding 1, this also determines normal and emergency current ratings for all windings.

transformers_read_max_tap() → float
Gets the active winding maximum tap in per-unit.

transformers_read_min_tap() → float
Gets the active winding minimum tap in per-unit.

transformers_read_r() → float
Gets the active winding resistance in %.

transformers_read_r_neut() → float
Gets the active winding neutral resistance [ohms] for wye connections. Set less than zero ungrounded wye.

transformers_read_tap() → float
Gets the active winding tap in per-unit.

transformers_read_x_neut() → float
Gets the active winding neutral reactance [ohms] for wye connections.

transformers_read_xhl() → float
Gets the percent reactance between windings 1 and 2, on winding 1 kVA base. Use for 2 winding or 3 winding transformers.

transformers_read_xht() → float
Gets the percent reactance between windings 1 and 3, on winding 1 kVA base. Use for 3 winding transformers only.

transformers_read_xlt() → float
Gets he percent reactance between windings 2 and 3, on winding 1 kVA base. Use for 3 winding transformers only.

transformers_write_kv(*argument*) → float
Sets the active winding kV rating. Phase-phase for 2 or 3 phases, actual winding kV 1 phase transformer.

transformers_write_kva (*argument*) → float
Sets the active winding kVA rating. On winding 1, this also determines normal and emergency current ratings for all windings.

transformers_write_max_tap (*argument*) → float
Sets the active winding maximum tap in per-unit.

transformers_write_min_tap (*argument*) → float
Sets the active winding minimum tap in per-unit.

transformers_write_r (*argument*) → float
Sets the active winding resistance in %.

transformers_write_r_neut (*argument*) → float
Sets the active winding neutral resistance [ohms] for wye connections. Set less than zero ungrounded wye.

transformers_write_tap (*argument*) → float
Sets the active winding tap in per-unit.

transformers_write_x_neut (*argument*) → float
Sets the active winding neutral reactance [ohms] for wye connections.

transformers_write_xhl (*argument*) → float
Sets the percent reactance between windings 1 and 2, on winding 1 kVA base. Use for 2 winding or 3 winding transformers.

transformers_write_xht (*argument*) → float
Sets the percent reactance between windings 1 and 3, on winding 1 kVA base. Use for 3 winding transformers only.

transformers_write_xlt (*argument*) → float
Sets the percent reactance between windings 2 and 3, on winding 1 kVA base. Use for 3 winding transformers only.

class py_dss_interface.models.Transformers.Transformers.**TransformersI** (*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/modify the properties of the Transformers Class where the values are integers.

The structure of the interface is as follows: int32_t TransformersI(int32_t Parameter, int32_t argument) ;

This interface returns an integer (signed 32 bits), the variable "parameter" is used to specify the property of the class to be used and the variable "argument" can be used to modify the value of the property when necessary. Reading and writing properties are separated and require a different parameter number to be executed.

The properties (parameter) are integer numbers and are described as follows.

transformers_count () → int
Gets the number of transformers within the active circuit.

transformers_first () → int
Sets the first Transformer active. Return 0 if no more.

transformers_next () → int
Sets the next Transformer active. Return 0 if no more.

transformers_read_is_delta () → int
Gets the information about if the active winding is delta (1) or wye (0) connection.

transformers_read_num_taps () → int
Gets the active winding number of tap steps between MinTap and MaxTap.

transformers_read_num_windings () → int
Gets the number of windings on this transformer. Allocates memory; set or change this property first.

transformers_read_wdg() → int
Gets the active winding number from 1..NumWindings. Update this before reading or setting a sequence of winding properties (R, Tap, kV, kVA, etc.).

transformers_write_is_delta(*argument*) → int
Sets the information about if the active winding is delta (1) or wye (0) connection.

transformers_write_num_taps(*argument*) → int
Sets the active winding number of tap steps between MinTap and MaxTap.

transformers_write_num_windings(*argument*) → int
Sets the number of windings on this transformer. Allocates memory; set or change this property first.

transformers_write_wdg(*argument*) → int
Sets the active winding number from 1..NumWindings. Update this before reading or setting a sequence of winding properties (R, Tap, kV, kVA, etc.).

class py_dss_interface.models.Transformers.Transformers.**TransformersS**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/modify the properties of the Transformers Class where the values are Strings.

The structure of the interface is as follows: CStr TransformersS(int32_t Parameter, CStr argument) ;

This interface returns a string, the variable "parameter" is used to specify the property of the class to be used and the variable "argument" can be used to modify the value of the property when necessary. Reading and writing properties are separated and require a different parameter number to be executed.

The properties (parameter) are integer numbers and are described as follows.

transformers_read_name()
Gets the active transformer name.

transformers_read_xfmr_code()
Gets the name of an XfrmCode that supplies electrical parameters for this transformer.

transformers_str_wdg_voltages()
Gets the voltages at the active winding of the active transformer in string format.

transformers_write_name(*argument*)
Sets the active transformer by name.

transformers_write_xfmr_code(*argument*)
Sets the name of an XfrmCode that supplies electrical parameters for this transformer.

class py_dss_interface.models.Transformers.Transformers.**TransformersV**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/modify the properties of the Transformers Class where the values are Variants.

The structure of the interface is as follows: void TransformersV(int32_t Parameter, VARIANT *Argument) ;

This interface returns a Variant, the variable "parameter" is used to specify the property of the class to be used and the variable "argument" can be used to modify the value of the property when necessary. Reading and writing properties are separated and require a different parameter number to be executed.

The properties (parameter) are integer numbers and are described as follows.

transformers_all_Names()
Gets a variant array of strings with all Transformer names in the active circuit.

transformers_wdg_currents()
Gets a a variant array of doubles containing the currents at the active winding on the active transformer. These currents come as complex pairs.

transformers_wdg_voltages()
Gets a variant array of doubles containing the voltages at the active winding on the active transformer. These voltages come as complex pairs.

4.1.38 VSources

Created by eniocc at 11/10/2020

class py_dss_interface.models.VSources.VSources.**VSources**(*obj_dss*)
Bases: py_dss_interface.models.VSources.VSourcesS.VSourcesS, py_dss_interface.models.VSources.VSourcesV.VSourcesV, py_dss_interface.models.VSources.VSourcesI.VSourcesI, py_dss_interface.models.VSources.VSourcesF.VSourcesF

This interface implements the Vsources (IVSources) interface of OpenDSS by declaring 4 procedures for accessing the different properties included in this interface: VSourcesS, VSourcesV, VSourcesI, VSourcesF.

class py_dss_interface.models.VSources.VSources.**VSourcesF**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: double VSourcesF(int32_t Parameter, double Argument);

This interface returns a floating point number with the result of the query according to the value of the variable Parameter, which can be one of the following.

vsources_read_angle_deg() → float
Gets the source phase angle of first phase in degrees.

vsources_read_base_kv() → float
Gets the source voltage in kV.

vsources_read_frequency() → float
Gets the source frequency in Hz.

vsources_read_pu() → float
Gets the source voltage in pu.

vsources_write_angle_deg(*argument*) → float
Sets the source phase angle of first phase in degrees.

vsources_write_base_kv(*argument*) → float
Sets the source voltage in kV.

vsources_write_frequency(*argument*) → float
Sets the source frequency in Hz.

vsources_write_pu(*argument*) → float
Sets the source voltage in pu.

class py_dss_interface.models.VSources.VSources.**VSourcesI**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: int32_t VSourcesI(int32_t Parameter, int32_t Argument);

This interface returns an integer number with the result of the query according to the value of the variable Parameter, which can be one of the following.

vsources_count() → int
Returns the number of VSource objects currently defined in the active circuit.

vsources_first() → int
Sets the first VSource to be active; returns 0 if none.

vsources_next() → int
Sets the next VSource to be active; returns 0 if none.

vsources_read_phases() → int
Gets the number of phases of the active VSource.

vsources_write_phases(*argument*) → int
Sets the number of phases of the active VSource.

class py_dss_interface.models.VSources.VSources.**VSourcesS**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: CStr VSourcesS(int32_t Parameter, CStr Argument);

This interface returns a string with the result of the query according to the value of the variable Parameter, which can be one of the following.

vsources_read_name()
Gets the name of the active VSource.

vsources_write_name(*argument*)
Sets the name of the active VSource.

class py_dss_interface.models.VSources.VSources.**VSourcesV**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: void VSourcesV(int32_t Parameter, VARIANT *Argument);

This interface returns a Variant with the result of the query according to the value of the variable Parameter, which can be one of the following.

vsources_all_names()
Gets the name of the active VSource.

4.1.39 XYCurves

Created by eniocc at 11/10/2020

class py_dss_interface.models.XYCurves.XYCurves.**XYCurves**(*obj_dss*)
Bases: py_dss_interface.models.XYCurves.XYCurvesS.XYCurvesS, py_dss_interface.models.XYCurves.XYCurvesI.XYCurvesI, py_dss_interface.models.XYCurves.XYCurvesF.XYCurvesF, py_dss_interface.models.XYCurves.XYCurvesV.XYCurvesV

This interface implements the XYCurves (IXYCurves) interface of OpenDSS by declaring 4 procedures for accessing the different properties included in this interface: XYCurvesS, XYCurvesI, XYCurvesF, XYCurvesV.

class py_dss_interface.models.XYCurves.XYCurves.**XYCurvesF**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: double XYCurvesF(int32_t Parameter, double Argument);

This interface returns a floating point number with the result of the query according to the value of the variable Parameter, which can be one of the following.

xycurves_read_x() → float
Gets the interpolated value after setting Y.

xycurves_read_x_scale() → float
Gets the factor to scale X values from original curve.

xycurves_read_x_shift() → float
Gets the amount to shift X value from original curve.

xycurves_read_y() → float
Gets the interpolated value after setting X.

xycurves_read_y_scale() → float
Gets the factor to scale Y values from original curve.

xycurves_read_y_shift() → float
Gets the amount to shift Y value from original curve.

xycurves_write_x(*argument*) → float
Sets the X value.

xycurves_write_x_scale(*argument*) → float
Sets the factor to scale X values from original curve.

xycurves_write_x_shift(*argument*) → float
Sets the amount to shift X value from original curve.

xycurves_write_y(*argument*) → float
Sets the Y value.

xycurves_write_y_scale(*argument*) → float
Sets the factor to scale Y values from original curve.

xycurves_write_y_shift(*argument*) → float
Sets the amount to shift Y value from original curve.

class py_dss_interface.models.XYCurves.XYCurves.**XYCurvesI**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: int32_t XYCurvesI(int32_t Parameter, int32_t Argument);

This interface returns an integer with the result of the query according to the value of the variable Parameter, which can be one of the following.

xycurves_count() → int
Gets number of XYCurves in active circuit.

xycurves_first() → int
Sets first XYCurves object active; returns 0 if none.

xycurves_next() → int
Sets next XYCurves object active; returns 0 if none.

xycurves_read_npts() → int
Gets the number of points in X-Y curve.

xycurves_write_npts(*argument*) → int
Sets the number of points in X-Y curve.

class py_dss_interface.models.XYCurves.XYCurves.**XYCurvesS**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: CStr XYCurvesS(int32_t Parameter, CStr Argument);

This interface returns a string with the result of the query according to the value of the variable Parameter, which can be one of the following.

xycurves_read_name()
Gets the name of the active XYCurve Object.

xycurves_write_name(*argument*)
Sets the name of the active XYCurve Object.

class py_dss_interface.models.XYCurves.XYCurves.**XYCurvesV**(*obj_dss*)
Bases: py_dss_interface.models.Base.Base

This interface can be used to read/write certain properties of the active DSS object.

The structure of the interface is as follows: void XYCurvesS(int32_t Parameter, VARIANT *Argument);

This interface returns a Variant with the result of the query according to the value of the variable Parameter, which can be one of the following.

xycurves_read_x_array()
Gets the X values as a variant array of doubles. Set Npts to max number expected if setting.

xycurves_read_y_array()
Gets the Y values as a variant array of doubles. Set Npts to max number expected if setting..

xycurves_write_x_array(*argument*)
Sets the X values as a variant array of doubles specified in Argument. Set Npts to max number expected if setting.

xycurves_write_y_array(*argument*)
Sets the Y values as a variant array of doubles specified in Argument. Set Npts to max number expected if setting.

CHAPTER 5

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

5.1 Bug reports

When reporting a bug please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

5.2 Documentation improvements

py_dss_interface could always use more documentation, whether as part of the official py_dss_interface docs, in docstrings, or even on the web in blog posts, articles, and such.

5.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/PauloRadatz/py_dss_interface/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that code contributions are welcome :)

CHAPTER 6

Authors

- Paulo Radatz - https://www.linkedin.com/in/pauloradatz/ and paulo.radatz@gmail.com
- Ênio Viana - https://www.linkedin.com/in/enioviana/ and eniocc@gmail.com
- Rodolfo Pilar Londero - https://www.linkedin.com/in/rodolfolondero/ and rodolfopl@gmail.com

CHAPTER 7

Changelog

7.1 1.1.0 (2021-10-10)

- Code refactored
- PEP 8 in some methods

7.2 1.0.0 (2021-01-21)

- Code refactored
- Works on Linux version of OpenDSS provided in the package. This version is from 2020.
- Tests included
- Methods renamed to satisfy PEP 8 – Style Guide for Python Code

7.3 0.1.0 (2021-01-21)

- Update OpenDSS version to: OpenDSS Version 9.2.0.1; License Status: Open

7.4 0.0.9 (2021-01-21)

- text method returns string values (Jouni request)
- OpenDSS exe 64bits included

7.5 0.0.8 (2020-12-10)

- Allowing run multiple instances of DDLL without problems with memory - Gustavo asked it
- Allowing run multiple OpenDSS' DLLs
- Update OpenDSS version to: OpenDSS Version 9.1.3.3 (64-bit build); License Status: Open

7.6 0.0.7 (2020-10-22)

- PVsystems updated.
- Update OpenDSS version to: OpenDSS Version 9.1.0.1 (64-bit build); License Status: Open.
- Allowing to write values into Variant methods.

7.7 0.0.4 (2020-08-17)

- DSSDLL class can receive OpenDSS folder in order to use the user own OpenDSS.

7.8 0.0.1 (2020-06-12)

- Integrating CI.

7.9 0.0.0 (2020-06-12)

- First release on PyPI.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

A

B

C

D

E

F

G

I

L

M

P

R

S

T

V

X