EDS Calculation Editor: PyEDS User Guide Extension

22.01.2019

Copyright Notice

The content of this manual, including text, illustrations, figures, and examples, are intended exclusively to describe the use and utilization of the EDS Analytics software as defined in this document. Due to many unforeseen variables that might occur, associated with specific uses or utilizations of the product, Transition Technologies – Advanced Solutions Sp. z o. o. cannot assume liability or responsibility for actual use based upon the information specified within this manual.

No patent liability is assumed by Transition Technologies – Advanced Solutions Sp. z o. o. with respect to the use of software described in this manual.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, including electronic, mechanical, photocopying, recording or otherwise without the prior express written permission of Emerson Process Management.

No part of this document may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, including electronic, mechanical, photocopying, recording or otherwise without the prior express written permission of Transition Technologies – Advanced Solutions Sp. z o. o.

The document is the property of and contains Proprietary Information owned by Transition Technologies – Advanced Solutions Sp. z o. o. It is transferred in confidence and trust, and the user agrees to treat this document in strict accordance with the terms and conditions of the agreement under which this document was provided.

This manual has been drafted and printed in Poland and is subject to change without notice.

TRANSITION TECHNOLOGIES -

ADVANCED SOLUTIONS SP. Z O.O.

55 Pawia Street, Warsaw 01-030

Phone: (+48) 022 331 80 20

Fax: (+48) 022 331 80 30

Mail: TTAS.info@tt.com.pl

Contents

CHAPTER 1

Introduction to PyEDS 2

IN THIS SECTION

1.1 PyEDS Synopsis

PyEds2 module provides Python language binding for Enterprise Data Server.

Its API allows to:

- read static and dynamic fields of process points from Online Server
- update values of process points in Online Server
- calculate thermodynamic functions

⚠ Each time a '↔' is used within the code, it marks a line that should not be broken. If using the copy-paste method, delete the character to avoid syntax error.

1.2 Requirements

UsingPyEDS2 module requires the following software to work properly:

- Latest Python 2 version, available for given platform:
  - on Windows this is the most recent official 2.x release from python.org/downloads/windows.

Note: *Note that the architecture (32-bit or 64-bit) of Python must match that of PyEds2.*

  - on Linux, the official Python 2 package provided by the operating system should be used.

Note: *Python 3 is not supported by PyEDS2.*

CHAPTER 2

Using PyEDS2 in Python applications

IN THIS SECTION

2.1 Installation

PyEDS 2 module is provided by the EDSApi package.

When choosing EDSApi version, make sure to match architecture of the package to that of Python interpreter. To check the architecture of Python interpreter, run the following script:

```
import platform

print(platform.architecture()[0])
```

The EDSApi package contains several APIs, but only the following files are required by PyEds2:

- PyEds2.dll and loggereds2.dll on Windows
- PyEds2.so and libloggereds2.so on Posix platforms

Python will be able to import the module as long as these files can be found in interpreter's search path.

2.2 Basic usage

To use the module, it needs to be imported first with the following statement:

```
import PyEds2
```

At the top level, the module provides a set of factory functions which create specific parts of the API.

- PyEds2.createLive() - create a connection object for EDS Online Server, see *2.3 Interacting with EDS Online Server on page 8* for more information.
- PyEds2.createThermodynamics() - create a wrapper for EDS thermodynamic functions, see *2.4 Calling thermodynamic functions on page 14* for more information.

See *4.1 API Reference on page 22*

2.3 Interacting with EDS Online Server

PyEds2 can connect to EDS Online Server and inspect/modify values of its process points.

To access this feature, create an instance of PyEds2_Live class by calling PyEds2.createLive():

```
import PyEds2

live = PyEds2.createLive()
```

2.3.1 Connecting to EDS Online Server

To use the PyEDS2_Live object, it must be connected to an EDS Online Server. The connection process is started by calling the PyEds2_Live.connect() method with a valid mode of operation and an address of EDS Online Server.

All EDS “agents” (applications connecting to Online Server) work in one or both of the following modes:

- client - can read values of static and dynamic point fields from Online Server, this mode is used by most EDS client applications
- source - can send values of dynamic point fields to Online Server, typically used in ZD feeders

To select a mode, pass **`True`** in the corresponding boolean argument: (**`asClient`** / **`asSource`**) of PyEds2_Live.connect().

The address of EDS Online Server (along with other connection parameters) is specified in the options argument. The argument should be a dictionary containing at least the following entries:

- '**`rhost`**' - address of EDS Online Server
- '**`rport`**' - port of EDS Online Server (typically *43000*)
- '**`dbfile`**' - path to a point cache file (in a writable directory)

Note that when the '**`dbfile`**' is unspecified, a default, platform-specific path will be used. On some platforms this default path points to a directory that is writable only by users with administrative rights. Therefore it is recommended to provide a user-specific path for the cache file.

If the point cache file cannot be created, the connection will still proceed. However, subsequent connections will be unable to reuse the point cache and will need more time to synchronize with Online Server.

When a connection is no longer needed, it should be closed with PyEds2_Live.shut() method. This should be the last method called on the PyEds2_Live object.

Example:

```
import PyEds2

live = PyEds2.createLive()

# connect as client, with point cache written in current directory
live.connect(True, False, {
   'rhost': 'localhost',
   'dbfile': 43000,
   'dbfile': './points.db'
})

# do more work with the client here...

# close the connection
live.shut()
```

Note that PyEds2_Live.connect() only starts the connection process and returns immediately. Before point data can be accessed, user of the API must confirm that connection is fully synchronized with Online Server.

2.3.2 Monitoring status of EDS Online Server connection

PyEds2_Live connection can be in one of several states:

- "**Disconnected**"
- "**Connecting**"
- "**Connected**"
- "**ProtoMismatch**"
- "**ShuttingDown**"

Most of PyEds2_Live methods will be successful only when the client has reached the **Connected** state.

Current connection state can be obtained from a call to PyEds2_Live.connInfo(). Note that PyEDS2 updates connection state in a background thread, so the value may change dynamically.

New connection objects always start in **Disconnected** state. Calling PyEds2_Live.connect() causes a transition to the **Connecting** status. At this stage, the client will download definition of all points from EDS Online Server. When the synchronization is successful, the status returned by PyEds2_Live.connInfo() will change to "**Connected**".

PyEDS provides a helper method - PyEds2_Live.waitTillSynced(), which blocks the calling thread until the connection is established. If the method returns normally, the client is **Connected** and local point data is synchronized with Online Server. Otherwise, a **RuntimeError** is raised and subsequent calls to most PyEds2_Live methods will fail.

Example:

```
import PyEds2

live = PyEds2.createLive()
live.connect(
# parameters omitted...
})

try:
   live.waitTillSynced()
   print('Connection has been synchronized')

except:
   pass

live.shut()
```

If the application using PyEDS2 should remain interactive while it waits for connection to EDS Online Server, then the call to PyEds2_Live.waitTillSynced() can be replaced with a loop that invokes PyEds2_Live.connInfo():

```
import PyEds2
import time

live = PyEds2.createLive()
live.connect(
   # parameters omitted...

})
connectionInfo = live.connInfo()
```

```
while connectionInfo['state'] != 'Connected':
   print('Status: %s, point count: %d' % (connectionInfo['state'],↔
   connectionInfo['pointCount']))
   time.sleep(1.0)
   connectionInfo = live.connInfo()

print('Connection has been synchronized')

live.shut()
```

Note: *The example program also displays the number of point definitions that have been synchronized with EDS Online Server.*

2.3.3 Inspecting process points on EDS Online Server

EDS process point is conceptually a dictionary of field names and their values.

Each type of point (e.g. **ANALOG**, **PACKED**, etc.) may provide a different set of fields. For example, only **ANALOG** points have **BB** and **TB** fields, while only **BINARY** points have **RD** and **SD**.

Points fields can be grouped into two categories:

- static

  Most point fields belong to this category. Their values rarely change. EDS clients automatically download updated values for these fields.

- dynamic

  Frequently changing fields belong to this category: point value, quality, timestamp, etc. Values of these fields are not automatically fetched from Online Server by EDS clients. See Receiving live point samples from EDS Online Server for more information on accessing this data.

This section deals with static point fields only.

To read fields of a process point, first make sure that the PyEds2_Live object is connected. Otherwise point data might be unavailable or stale.

Then invoke PyEds2_Live.pointStaticInfo() method with identifier of the process point:

```
import PyEds2
import time

# connection setup omitted...

pointFields = live.pointStaticInfo('example_point')
print('Static point fields:')
print(' - type:', pointFields['RT'])
print(' - numeric identifier:', pointFields['LID'])
print(' - description:', pointFields['DESC'])
print(' - time of last static field change:', time.ctime(pointFields↔
['DTS'][0]))
```

The identifier passed to PyEds2_Live.pointStaticInfo() can be a string or a number. Strings are interpreted as an **IESS**, while numeric values are treated as a **LID**.

Point identifiers must be obtained from an external source (e.g. EDS Terminal), as PyEDS2 does not provide a function to enumerate all available points on EDS Online Server.

For a full list of supported static point fields, see the description of PyEds2_Live.pointStaticInfo().

2.3.4 Receiving live point samples from EDS Online Server

Note: *Live point samples can be read only by PyEds2_Live objects which have been initialized as client. Before attempting to use this feature, ensure that client mode was enabled when PyEds2_Live.connect() was called.*

Dynamic fields of EDS process points can change their value every second (although most points will typically be updated at a lower frequency). Mirroring all point values from EDS Online Server to each connected client could quickly saturate available network bandwidth and cause high CPU usage on the server.

To improve scalability, EDS implements a 'publish-subscribe' pattern for sending frequently updated point data to clients. Applications that wish to receive notifications when a dynamic field of a point changes, must first register themselves as subscribers of that point.

To accomplish that applications must call PyEds2_Live.subscribe() method, passing the identifier of requested point as parameter. The method accepts strings (which are treated as **`IESS`**) and numeric values (interpreted as a **`LID`**).

Note that the value of subscribed point does not become immediately available to the client. The PyEds2_Live.subscribe() function is asynchronous and does not wait for the subscription to complete.

To ensure that the client has received values for all recently subscribed points, call PyEds2_Live.waitTillSynced(). Applications that do not wish to block their thread are allowed to omit this call. In this case, a read of a dynamic point value while its subscription is still being processed will return a dummy value (with zeroes in most fields).

To obtain a dynamic value of a subscribed process point, invoke PyEds2_Live.pointDynamicInfo(), passing the identifier of that point as parameter. Both identifier forms (**`IESS`** and **`LID`**) are supported by the method.

When a subscription is no longer needed, remove it by calling PyEds2_Live.unsubscribe().

Example:

```
import PyEDS2
import time

SAMPLE_IESS = 'example_point'
SAMPLE_LID = 1234

live = PyEds2.createLive()

# connection setup omitted...

# subscribe a point using IESS
live.subscribe(SAMPLE_IESS)

# subscribe a point using LID
live.subscribe(SAMPLE_LID)

# wait for values of all subscribed points to become available
```

```
# this step is optional if receiving a dummy value from↔
`pointDynamicInfo()` is acceptable
live.waitTillSynced()

# fetch 10 samples point samples at 1 second interval
for _ in range(10):
   print('--------');
   print(live.pointDynamicInfo(SAMPLE_IESS))
   print(live.pointDynamicInfo(SAMPLE_LID))
   time.sleep(1.0)

# unsubscribe points (not required if `shut()` is immediately called)
live.unsubscribe(SAMPLE_IESS)
live.unsubscribe(SAMPLE_LID)

live.shut()
```

For a full list of supported dynamic point fields, see the description of PyEds2_Live.pointDynamicInfo().

2.3.5 Sending live point samples to EDS Online Server

> ***Note:*** *Live point samples can be sent only by PyEds2_Live objects which have been initialized as source. Before attempting to use this feature, ensure that source mode was enabled when PyEds2_Live.connect() was called.*

PyEDS2 module can send dynamic values of a process point to EDS Online Server. Applications using PyEDS2 in this mode operate like a simple ZD feeder.

To send live point values to EDS Online Server, pass **`True`** in the **`asSource`**argument of PyEds2_Live.connect() method. If all written points belong to the same **`ZD`**, it is recommended to specify it in the **`zd`** item of the **`options`** dictionary.

When connection to EDS Online Server is successfully established, the application should call PyEds2_Live.originate() for each point that is going to receive live values. Then it should assign a value to all originated points using PyEds2_Live.write(). The first assignment of point value should happen as soon as possible after origination. Otherwise, after several seconds, EDS Online Server will automatically reset the point to a value of 0 with timed-out status.

Once at least one value has been written to originated point, EDS Online Server will automatically start prolonging that sample in absence of further PyEds2_Live.write() calls. That is, as long as the "feeder" application does not call PyEds2_Live.unoriginate() or closes connection to Online Server, originated points will not be marked as timed out, even if no new values are written to them.

For each point, EDS Online Server will accept at most one sample per second. Values written at a higher frequency will be dropped.

The sample parameter passed to PyEds2_Live.write() can be either a tuple with three elements (**`value`**, **`timestamp`**, **`quality`**) or a numeric value (in which case current time and good quality will be assumed).

When application no longer wants to assign values to a point, it should call PyEds2_Live.unoriginate(). EDS Online Server will eventually mark such point as timed out.

Example:

```
import PyEds2
import random
import time

SAMPLE_IESS = 'example_point'
SAMPLE_LID = 1234

# connection setup omitted...

# originate a point using IESS
live.originate(SAMPLE_IESS)

# originate a point using LID
live.originate(SAMPLE_LID)

# assign 100 random samples to points at 1 second interval
for _ in range(100):
   value = random.random()
   random_quality = random.choice(['G', 'F', 'P', 'B'])

   # write point value with 1 hour shift and random quality


   live.write(SAMPLE_IESS, (value, time.time() - 3600,↔
   random_quality))
   # write point value with current time and good quality
   live.write(SAMPLE_LID, value)

   time.sleep(1.0)

# unoriginate points (not required if `shut()` is↔
immediately called)
live.unoriginate(SAMPLE_IESS)
live.unoriginate(SAMPLE_LID)

live.shut()
```

2.4 Calling thermodynamic functions

PyEDS2 provides a set of thermodynamic functions. To access them, create an instance of PyEds2.PyEds2_Thermodynamics class by calling PyEds2.createThermodynamics():

```
import PyEds2

td = PyEds2.createThermodynamics()

# display all available thermodynamic functions
dir(td)
```

Methods of PyEds2.PyEds2_Thermodynamics do not require a connection to Online Server.

In general they expect one or two arguments, and return a tuple (**value**, **quality**):

```
# calculate entropy of water at 82 °C
print(td.sslt(82.0))
```

2.5 Configuring internal EDS logger

PyEds2 comes with a builtin logger, that can write diagnostic messages to console (standard output) and text files.

The following aspects of logging are configurable:

- severity of logged messages (**debug=** parameter)
- destination of logged messages (**logger=** parameter)
- subset of client components allowed to log (**subsystems=** parameter, typically ALL)

It is recommended to log messages of at least error level (**debug=3**) severity. Availability of such log files can help troubleshoot issues.

To initialize logger, call PyEds2.setupLogger(). For example:

```
import PyEds2

# log errors to /var/log/pyeds2.log and 10 extra rotated files, with ↔
maximum size of 8 MB each
PyEds2.setupLogger('debug=3 logger=rfile:3,fileName:/var/log/↔
pyeds2.log,fileSizeLimit:8000000,fileCntLimit:10 subsystems=ALL')
```

For development, logging to standard output might be more convenient:

```
# log everything (up to 'debug' level) to console
PyEds2.setupLogger('debug=7 logger=console:7 subsystems=ALL')
```

See PyEds2.setupLogger() documentation for more information.

CHAPTER 3

Using PyEDS2 in EDS Calculation Server and Editor

IN THIS SECTION

3.1 Introduction

In addition to standalone applications, PyEDS2 module can also be used in EDS Calculation Server and Editor. In this context, PyEDS2 provides a preconfigured connection to Online Server and a set of thermodynamic functions. The feature can be accessed by any Python script that is called from a CDF file.

This chapter assumes that the reader is familiar with the interface of PyEDS2 module. For more information on the topic, consult *3.4 Using PyEds2 in Python calculation modules on page 19*.

3.2 Installation

EDSSrv and EDSCalc packages already contain the PyEDS2 module.

However, to use Python scripts in EDS calculation files (CDFs), the user must also install the CPython interpreter. See the *1.2 Requirements on page 6* for more information.

3.3 Calling Python functions from a calculation script

Before PyEDS2 module can be used in EDS calculations, the CDF file must first include a Python script and call at least one function from it.

Assuming that the Python script will be named **`example.py`**, add the following line at the top of the CDF file:

```
include 'example.py'

# rest of the CDF file...
```

Then define a calculation statement which calls a Python function (named **`pysum`** in this example). Here we assume that the CDF file contains variables **`x`**, **`y`** and **`result`** (possibly mapped to a set of process points):

```
result := 'example.py'@pysum(x, y);
```

Finally, create the **`example.py`** script in the same directory as the calling CDF file.

The **`pysum`** function must match exactly the signature of the call from the CDF calculation file (i.e. have two arguments and return a value):

```
# sum values of `x` and `y`
def pysum(x, y):
   return (x[0], x[1] + y[1], x[2], x[3], 0, '')
```

The type of each argument and the returned value is a tuple containing 6 items:

- point's type ('**`A`**', '**`B`**', '**`D`**', '**`I`**', or '**`P`**')
- value (**`float`** for ANALOG and DOUBLE points, **`int`** for BINARY, PACKED and INT64)
- quality ('**`G`**', '**`F`**', '**`P`**', '**`B`**', '**`N`**' or '**`O`**')
- timestamp (number of seconds since 1 January 1970)
- point's SID (numeric identifier of the point)
- point's IESS

Note that the returned value does not need to contain a valid SID and IESS.

EDS will now call the **`pysum`** function whenever the value of x or y variable in the CDF file changes.

3.4 Using PyEds2 in Python calculation modules

The first step required to use PyEDS2 in a standalone Python application is to import the **PyEds2** module and create an instance of PyEds2_Live or PyEds2_Thermodynamics, as described in *2 Using PyEDS2 in Python applications on page 7*.

However, when using PyEDS2 from EDS calculation engine, neither of these steps is required. During startup, EDS Calculation Server and Editor automatically imports the module and creates an instance of each PyEds2 class.

Python scripts called from CDF files need only to obtain a reference to these premade PyEds2 objects. To accomplish that, a special function named **_calc_init** must be defined in the Python file. When calculations are started, the function will be called with two parameters:

- a PyEds2_Live instance, connected to EDS Online Server in client and source mode
- a PyEds2_Thermodynamics object

To Python module is free to store references to these PyEds2 objects and use them afterwards in other functions (such as those called from CDF files):

```
live = None
thermodynamics = None

def _calc_init(live_, thermodynamics_):
   global live, thermodynamics

   live = live_
   thermodynamics = thermodynamics_
   live.subscribe('extra_point')
   live.waitTillSynced()

# sum values of `x`, `y` and \extra_point\
def pysum(x, y):
   extra_point_sample = live.pointDynamicInfo('extra_point')
   sum = x[1] + y[1] + extra_point_sample['AV']

   return (x[0], sum, x[2], x[3], 0, '')
```

3.5 Troubleshooting

3.5.1 Arguments passed to the _calc_init function have None value

Arguments to **_calc_init** have the value of **None** when PyEds2 module can't be loaded by the calculation engine. Confirm that the PyEds2 dynamic library can be found in the installation directory of EDS Calculation Server / Editor.

CHAPTER 4

API Reference

IN THIS SECTION

4.1 API Reference

`PyEds2.createLive()`

Return a new PyEds2_Live instance.

`PyEds2.createThermodynamics()`

Return a new PyEDS2_Thermodynamics instance.

`PyEds2.setupLogger(options)`

Set logger options.

`options` is a string containing **`key=value`** pairs separated with spaces. The following keys are recognized:

- **`debug`** - maximum numeric level of logged messages, with 1 being the most severe and 7 being the least significant
- **`logger`** - destination of logged messages

  Commonly used loggers:

  - **`console`**

  For example:

  ```
  logger=console:3
  ```

  will display messages with severity level 1-3 in the console used to start the application

  - **`rfile`** (rotated files)

  For example:

```
logger=rfile:5,fileName:/home/user/.eds/pyeds.log,↔
fileSizeLimit:10000000,fileCntLimit:3
```

  will write messages with severity level 1-5 to a file with pyeds.log prefix, until it reaches the size of 10 MB. Then a numeric suffix will be added to the log file and an empty pyeds.log file will be created for new messages. Up to 3 files with old messages will be kept.

  It is recommended to specify the location of destination file using an absolute (full) path.

- **`subsystems`** - which subsystems of the application should be allowed to log messages, **`ALL`** by default

Example invocation:

```
log_path = os.path.abspath('api.log')
options = 'debug=7

logger=rfile:6,fileName:%s,fileSizeLimit:5000000,fileCntLimit:10↔
subsystems=ALL' % log_path
PyEds2.setupLogger(options)
```

4.2 PyEds2_Live

`PyEds2.PyEds2_Live`

Provide access to EDS Online Server.

Call PyEds2_Live to obtain an instance of this class.

connInfo()

```
connInfo()
```

Return dictionary with information about current connection.

The dictionary will contain entries for the following keys:

- "**state**" - string - current connection state - one of the following values:
  - "**Disconnected**"
  - "**Connecting**"
  - "**Connected**"
  - "**ProtoMismatch**"
  - "**ShuttingDown**"
- "**pointCount**" - integer - number of process points provided by Online Server
- "**rhost**" - string - Online Server's address
- "**rport**" - integer - Online Server's listening port
- "**serverTime**" - tuple - if connected, current Online Server's time as a tuple, otherwise **None**.

The time tuple has two elements:

- number of seconds since 1 January 1970
- number of microseconds since last full second

This method is safe to call on an unconnected PyEds2_Live object.

Raises **RuntimeError** on invalid arguments.

connect()

```
connect (asClient, asSource, options)
```

Initialize connection to Online Server.

asClient and **asSource** are boolean arguments. At least one of them must be **True**.

- When **asClient** is **True**, the connection will periodically download values of all points registered with subscribe() method.
- When **asSource** is **True**, the connection will periodically upload values of all points registered with originate() method.

  options is a dictionary with the following keys supported:

  - "**lhost**" - name of local host to use for connection to Online Server (specify "**0.0.0.0**" for any host)
  - "**lhost**" - local port to use for connection to Online Server (specify **0** for any port)
  - "**lport_range**" - range of local ports to use for connection to Online Server (**PyEds** will scan ports from **lport** to **lport+lport_range** until it finds one that is unbound)
  - "**rhost**" - host of the Online Server
  - "**rport**" - port of the Online Server
  - "**zd**" - ZD to use for connection
  - "**comm_timeout**" - communication timeout in seconds
  - "**resp_timeout**" - response timeout in seconds
  - "**timeout**" - agent timeout in seconds
  - "**compress**" - if **True**, enable compression of network packets
  - "**max_packet**" - maximum allowed size of network packet
  - "**dbfile**" - path where a cache file with point data will be created (location must be writable)

  Most of the keys in the **options** dictionary can be omitted, which will cause **PyEds** to use default values.

  This is typically the method that should be called first on a PyEds2_Live object. Unless specifically noted otherwise, PyEds2_Live methods will fail when called on an unconnected object.

  Note that connect() only initializes the client and does not wait until the connection is fully synchronized. Call waitTillSynced() to make sure that connection is ready.

  Raises **RuntimeError** on invalid arguments or connection failure.

originate()

originate(pointID)

Start sending point values to Online Server.

pointId must be a valid point identifier. If **pointId** is a string, it is interpreted as IESS. Otherwise the argument is converted to an integer and treated as LID.

It is allowed to call originate() more than once with the same **pointId**. However, EDS client does not count how many times a point has been originated. The first call to unoriginate() will stop sending value updates for that point.

The point must not be already subscribed by the same PyEds2_Live object.

To send values of originated points to Online Server, use the write() method.

Call unoriginate() to stop sending values for the point.

Raises **RuntimeError** on invalid arguments, when connection has not been initialized as source in connect(), or when the specified point is already subscribed.

pointDynamicInfo()

pointDynamicInfo(pointId)

Get values of dynamic point fields.

pointId should be a valid point identifier. If **pointId** is a string, it is interpreted as IESS. Otherwise the argument is converted to an integer and treated as LID.

The point should be subscribed with subscribe() before calling pointDynamicInfo().

The result of the function is a dictionary with the following keys:

- "**QUAL**" - character - point's quality ('**G**', '**F**', '**P**', '**B**', '**N**' or '**O**')
- "**ST**" - integer - point's status
- "**TS**" - tuple - point's timestamp
- "**TSS**" - integer - point's timestamp shift
- "**AT**" - tuple - point's alarm timestamp
- "**RT**" - character - points's type:
  - "**A**" - ANALOG
  - "**B**" - BINARY
  - "**D**" - DOUBLE
  - "**I**" - INT64
  - "**P**" - PACKED
- "**AV**" - float - point's value, only present for ANALOG points
- "**BV**" - integer - point's value, only present for BINARY points
- "**DV**" - float - point's value, only present for DOUBLE points
- "**IPV**" - long - point's value, only present for INT64 points
- "**PV**" - integer - point's value, only present for PACKED points

  The time tuple returned for "TS" and "AT" keys has two elements:

  - number of seconds since 1 January 1970
  - number of microseconds since last full second

  When called with an invalid **pointId** or on unsynchronized connection, pointDynamicInfo() will return dummy point information in the dictionary.

  Raises **RuntimeError** on invalid arguments.

pointStaticInfo()

pointStaticInfo(pointId)

Get values of static point fields.

pointId should be a valid point identifier. If **pointId** is a string, it is interpreted as IESS. Otherwise the argument is converted to an integer and treated as LID.

The result of the function is a dictionary with the following keys:

- "**SID**" - integer - point's server-side id
- "**LID**" - integer - point's client-side id
- "**DF**" - integer - point's DF flags
- "**IESS**" - string - point's IESS
- "**IDCS**" - string - point's IDCS
- "**DESC**" - string - point's description
- "**ZD**" - string - point's ZD
- "**AR**" - character - point's archive mode
- "**AUX**" - string - point's auxiliary data
- "**UN**" - string - point's unit
- "**DTS**" - tuple - timestamp of point's most recent static field change
- "**RT**" - character - point's type:
  - "**A**" - ANALOG
  - "**B**" - BINARY
  - "**D**" - DOUBLE
  - "**I**" - INT64
  - "**P**" - PACKED
- "**BB**" - float - point's bottom bar value, only present for ANALOG points
- "**TB**" - float - point's top bar value, only present for ANALOG points
- "**LL**" - float - point's low limit value, only present for ANALOG points
- "**HL**" - float - point's high limit value, only present for ANALOG points
- "**RD**" - string - point's reset state description, only present for BINARY points
- "**SD**" - string - point's set state description, only present for BINARY points
- "**DBB**" - float - point's bottom bar value, only present for DOUBLE points
- "**DTB**" - float - point's top bar value, only present for DOUBLE points
- "**DLL**" - float - point's low limit value, only present for DOUBLE points
- "**DHL**" - float - point's high limit value, only present for DOUBLE points

The time tuple returned for "**DTS**" key has two elements:

- number of seconds since 1 January 1970
- number of microseconds since last full second

When called with an invalid **pointId** or on unsynchronized connection, pointStaticInfo() will return dummy point information in the dictionary.

Raises **RuntimeError** on invalid arguments.

shut()

```
shut()
```

Close connection to Online Server.

Raises **RuntimeError** on invalid arguments or when connection is already being shut down.

subscribe()

```
subscribe(pointId)
```

Start receiving point values from Online Server.

pointId must be a valid point identifier. If **pointId** is a string, it is interpreted as IESS. Otherwise the argument is converted to an integer and treated as LID.

It is allowed to call subscribe() more than once with the same **pointId**. EDS client internally maintains a subscription counter for each point.

The point must *not* be already originated by the same PyEds2.PyEds2_Live object.

To read values of subscribed points, use the pointDynamicInfo() method.

Call unsubscribe() to stop receiving values for the point.

Raises **RuntimeError** on invalid arguments, when connection has not been initialized as client in connect(), or when the specified point is already originated.

unoriginate()

```
unoriginate(pointId)
```

Stop sending point values to Online Server.

pointId must be a valid point identifier. If **pointId** is a string, it is interpreted as IESS. Otherwise the argument is converted to an integer and treated as LID.

EDS client does not count how many times a point has been originated. The first call to unoriginate() will stop sending value updates for that point.

Raises **RuntimeError** on invalid arguments or when connection has not been initialized as source in connect().

unsubscribe()

```
unsubscribe(pointId)
```

Stop receiving point values from Online Server.

pointId must be a valid point identifier. If **pointId** is a string, it is interpreted as IESS. Otherwise the argument is converted to an integer and treated as LID.

If a point was subscribed more than once, then unsubscribe() must be called the same number of times to actually remove the subscription. EDS client internally maintains a subscription count for each point and stops receiving point updates only when the counter value reaches 0.

Raises **RuntimeError** on invalid arguments or when connection has not been initialized as client in connect().

waitTillSynced()

```
waitTillSynced()
```

Wait until connection is synchronized.

The method will block until the connection is fully established or an unrecoverable error occurs.

It can also be used after a call to PyEDS2_Livesubscribe(), to ensure that values of recently subscribed points have been received by the client.

Raises **RuntimeError** on invalid arguments or when connection cannot be established.

write()

write(pointId, sample)

Write value to a process point on Online Server.

pointId should be a valid point identifier. If **pointId** is a string, it is interpreted as IESS. Otherwise the argument is converted to an integer and treated as LID.

sample argument can be a numeric type (float, integer) or a tuple.

If **sample** is a tuple, then it should have the following elements: (**value**, **timestamp**, [**quality**]), where:

- **value** - integer/float - value written to process point
- **timestamp** - integer - number of seconds since 1 January 1970
- **quality** - character - optional quality of the value ('**G**', '**F**', '**P**', or '**B**')

If sample is a numeric value, then it is converted to the following tuple: (**sample**, **time.time()**, '**G**').

The recommended method of sending point values is to initialize PyEds2_Live as source and call originate() for each point that needs to be written. However, even in client mode, PyEds2_Live can assign values to process points, albeit in a much less efficient way. In addition, the following constraints must be met to write a point value in client mode:

- the **DF_ALLOW_OPERATE** flag must be set in point's **DF** field
- the point must have at least one of SG control groups (in range 192-255) set

Raises **RuntimeError** on invalid arguments or when connection is initialized as client and the point does not have the **DF_ALLOW_OPERATE**flag set.

4.3 PyEDS2_Thermodynamics

PyEDS2_Thermodynamics provides access to EDS thermodynamic functions.

Call PyEDS2_Thermodynamics to obtain an instance of this class.

asssp(enthalpy, pressure)

Return speed of sound [m/s] in superheated steam in the function of enthalpy and pressure.

The result is a tuple containing calculated function value and its quality ('**G**', '**F**', '**P**' or '**B**').

ccltp(temperature, pressure)

Return heat capacity [kJ/(kg.°C)] of water in the function of temperature and pressure.

The result is a tuple containing calculated function value and its quality ('**G**', '**F**', '**P**' or '**B**').

csstp(temperature, pressure)

Return heat capacity [kJ/(kg.°C)] (at constant pressure) of superheated steam in the function of temperature and pressure.

The result is a tuple containing calculated function value and its quality ('G', 'F', 'P' or 'B').

hcltp(temperature, pressure)

Return enthalpy [kJ/kg] of water in the function of temperature and pressure.

The result is a tuple containing calculated function value and its quality ('G', 'F', 'P' or 'B').

hslp(pressure)

Return enthalpy [kJ/kg] of saturated water in the function of pressure.

The result is a tuple containing calculated function value and its quality ('G', 'F', 'P' or 'B').

hslt(temperature)

Return enthalpy [kJ/kg] of saturated water in the function of temperature.

The result is a tuple containing calculated function value and its quality ('G', 'F', 'P' or 'B').

hsssp(entropy, pressure)

Return enthalpy [kJ/kg] of superheated steam in the function of entropy and pressure.

The result is a tuple containing calculated function value and its quality ('G', 'F', 'P' or 'B').

hsstp(temperature, pressure)

Return enthalpy [kJ/kg] of superheated steam in the function of temperature and pressure.

The result is a tuple containing calculated function value and its quality ('G', 'F', 'P' or 'B').

hsvp(pressure)

Return enthalpy [kJ/kg] of saturated steam in the function of pressure.

The result is a tuple containing calculated function value and its quality ('G', 'F', 'P' or 'B').

hsvp1(pressure)

Return enthalpy [kJ/kg] of saturated steam in the function of pressure.

The result is a tuple containing calculated function value and its quality ('G', 'F', 'P' or 'B').

hsvt(temperature)

Return enthalpy [kJ/kg] of saturated steam in the function of temperature.

The result is a tuple containing calculated function value and its quality ('G', 'F', 'P' or 'B').

hwssp(entropy, pressure)

Return enthalpy [kJ/kg] of wet steam in the function of entropy and pressure.

The result is a tuple containing calculated function value and its quality ('G', 'F', 'P' or 'B').

hwssp1(entropy, pressure)

Return enthalpy [kJ/kg] of wet steam in the function of entropy and pressure.

The result is a tuple containing calculated function value and its quality ('G', 'F', 'P' or 'B').

hwsts(temperature, entropy)

Return enthalpy [kJ/kg] of wet steam in the function of temperature and entropy.

The result is a tuple containing calculated function value and its quality ('G', 'F', 'P' or 'B').

lcltp()

lsstp()

psatt(temperature)

Return pressure [bar] of saturated steam in the function of temperature.

The result is a tuple containing calculated function value and its quality ('G', 'F', 'P' or 'B').

psshs(enthalpy, pressure)

Return pressure [bar] of superheated steam in the function of enthalpy and pressure.

The result is a tuple containing calculated function value and its quality ('G', 'F', 'P' or 'B').

pwshs(enthalpy, entropy)

Return pressure [bar] of wet steam in the function of enthalpy and entropy.

The result is a tuple containing calculated function value and its quality ('G', 'F', 'P' or 'B').

pwshs1(enthalpy, entropy)

Return pressure [bar] of wet steam in the function of enthalpy and entropy.

The result is a tuple containing calculated function value and its quality ('G', 'F', 'P' or 'B').

scltp(temperature, pressure)

Return entropy [kJ/(kg.°C)] of water in the function of temperature and pressure.

The result is a tuple containing calculated function value and its quality ('G', 'F', 'P' or 'B').

sslt(temperature)

Return entropy [kJ/(kg.°C)] of saturated water in the function of temperature.

The result is a tuple containing calculated function value and its quality ('G', 'F', 'P' or 'B').

ssshp(enthalpy, pressure)

Return entropy [kJ/(kg.°C)] of superheated steam in the function of enthalpy and pressure.

The result is a tuple containing calculated function value and its quality ('G', 'F', 'P' or 'B').

ssstp(temperature, pressure)

Return entropy [kJ/(kg.°C)] of superheated steam in the function of temperature and pressure.

The result is a tuple containing calculated function value and its quality ('G', 'F', 'P' or 'B').

ssvp(pressure)

Return entropy [kJ/(kg.°C)] of saturated steam in the function of pressure.

The result is a tuple containing calculated function value and its quality ('G', 'F', 'P' or 'B').

ssvt(temperature)

Return entropy [kJ/(kg.°C)] of saturated steam in the function of temperature.

The result is a tuple containing calculated function value and its quality ('G', 'F', 'P' or 'B').

swshp(enthalpy, pressure)

Return entropy [kJ/(kg.°C)] of wet steam in the function of enthalpy and pressure.

The result is a tuple containing calculated function value and its quality ('G', 'F', 'P' or 'B').

swshp1(enthalpy, pressure)

Return entropy [kJ/(kg.°C)] of wet steam in the function of enthalpy and pressure.

The result is a tuple containing calculated function value and its quality ('G', 'F', 'P' or 'B').

tclhp(enthalpy, pressure)

Return temperature [°C] of water in the function of enthalpy and pressure.

The result is a tuple containing calculated function value and its quality ('G', 'F', 'P' or 'B').

tclsp(entropy, pressure)

Return temperature [°C] of water in the function of entropy and pressure.

The result is a tuple containing calculated function value and its quality ('G', 'F', 'P' or 'B').

tsatp(pressure)

Return temperature [°C] of saturated steam in the function of pressure.

The result is a tuple containing calculated function value and its quality ('G', 'F', 'P' or 'B').

tsshp(enthalpy, pressure)

Return temperature [°C] of superheated steam in the function of enthalpy and pressure.

The result is a tuple containing calculated function value and its quality ('G', 'F', 'P' or 'B').

tsssp(entropy, pressure)

Return temperature [°C] of superheated steam in the function of entropy and pressure.

The result is a tuple containing calculated function value and its quality ('G', 'F', 'P' or 'B').

twshp(enthalpy, pressure)

Return temperature [°C] of wet steam in the function of enthalpy and pressure.

The result is a tuple containing calculated function value and its quality ('G', 'F', 'P' or 'B').

twshp1(enthalpy, pressure)

Return temperature [°C] of wet steam in the function of enthalpy and pressure.

The result is a tuple containing calculated function value and its quality ('G', 'F', 'P' or 'B').

twshs(enthalpy, entropy)

Return temperature [°C] of wet steam in the function of enthalpy and entropy.

The result is a tuple containing calculated function value and its quality ('G', 'F', 'P' or 'B').

twssp(entropy, pressure)

Return temperature [°C] of wet steam in the function of entropy and pressure.

The result is a tuple containing calculated function value and its quality ('G', 'F', 'P' or 'B').

twssp1(entropy, pressure)

Return temperature [°C] of wet steam in the function of entropy and pressure.

The result is a tuple containing calculated function value and its quality ('G', 'F', 'P' or 'B').

vcltp(temperature, pressure)

Return real volume [m3/kg] of water in the function of temperature and pressure.

The result is a tuple containing calculated function value and its quality ('G', 'F', 'P' or 'B').

vslt(temperature)

Return real volume [m3/kg] of saturated water in the function of temperature.

The result is a tuple containing calculated function value and its quality ('G', 'F', 'P' or 'B').

vsshp(enthalpy, pressure)

Return real volume [m3/kg] of superheated steam in the function of enthalpy and pressure.

The result is a tuple containing calculated function value and its quality ('G', 'F', 'P' or 'B').

vsshs(enthalpy, entropy)

Return real volume [m3/kg] of superheated steam in the function of enthalpy and entropy.

The result is a tuple containing calculated function value and its quality ('G', 'F', 'P' or 'B').

vsssp(entropy, pressure)

Return real volume [m3/kg] in superheated steam in the function of entropy and pressure.

The result is a tuple containing calculated function value and its quality ('G', 'F', 'P' or 'B').

vsstp(temperature, pressure)

Return real volume [m3/kg] of superheated steam in the function of temperature and pressure.

The result is a tuple containing calculated function value and its quality ('G', 'F', 'P' or 'B').

vsvp(pressure)

Return real volume [m3/kg] of saturated steam in the function of pressure.

The result is a tuple containing calculated function value and its quality ('G', 'F', 'P' or 'B').

vwshp(enthalpy, pressure)

Return real volume [m3/kg] of wet steam in the function of enthalpy and pressure.

The result is a tuple containing calculated function value and its quality ('G', 'F', 'P' or 'B').

vwshs(enthalpy, entropy)

Return real volume [m3/kg] of wet steam in the function of enthalpy and entropy.

The result is a tuple containing calculated function value and its quality ('G', 'F', 'P' or 'B').

vwshs1(enthalpy, entropy)

Return real volume [m3/kg] of wet steam in the function of enthalpy and entropy.

The result is a tuple containing calculated function value and its quality ('G', 'F', 'P' or 'B').