PyPlcnextRsc

Release 0.1.7

Song Yantao (PXC/IMA&STE)

Aug 30, 2021

CONTENTS

Warning: This project is under development stage, please always get the latest version at https://gitlab.phoenixcontact.com/SongYantao/pyplcnextrsc

PyPLCnextRsc is a simple ,yet elegant, RSC client library. it allows you to use RSC service extremely easily.

Have a look at *the gallery* to get an idea of what is possible.

CHAPTER

ONE

A SIMPLE CLIENT

The following snippet is the most simple example, it uses the *IPlcManagerService* to `Cold Start` the PLCnext

```
from PyPlcnextRsc import *
from PyPlcnextRsc.Arp.Plc.Domain.Services import IPlcManagerService, PlcStartKind

if __name__ == "__main__":
    with Device('192.168.1.10', secureInfoSupplier=GUISupplierExample) as device:
        plc_manager_service = IPlcManagerService(device)  # Get PlcManagerService
        plc_manager_service.Stop()
        plc_manager_service.Start(PlcStartKind.Cold)
```

Ready to start learning ? *Click*

CHAPTER

TWO

CONTENTS

2.1 Gallery

Showcase, demonstrating the possibilities of PyPlcnextRsc.

coming soon

2.2 Getting started

2.2.1 Installation

Requires: **Python>=3.7.6**

```
$ pip install -U PyPlcnextRsc
```

Warning: This project is under development stage, please always get the latest version at https://gitlab.phoenixcontact.com/SongYantao/pyplcnextrsc

2.2.2 Create connection

In order to use RSC service , you need to get an already connected **Device** instance. see *Device*

To do this , there are two code-styles :

```
from PyPlcnextRsc import Device, GUISupplierExample

# 1st. Use with-block , which will automatically connect and dispose the connection
with Device('192.168.1.10', secureInfoSupplier=GUISupplierExample) as device:
    ...  # do rsc operation
```

(continues on next page)

(continued from previous page)

```
# ------------------------------------------------------------------------------
# 2nd. Use traditional way , which you should connect and dispose the connection␣
↪explicitly

device = Device('192.168.1.10', secureInfoSupplier=GUISupplierExample)
device.connect()
...  # do rsc operation
device.dispose()
```

If PLCnect's *User Authentication* is disabled , just ignore *secureInfoSupplier* , like this:

```
with Device('192.168.1.10') as device:
    ...  # do rsc operation
```

Note: If running the package locally , use IP *127.0.0.1*

This package use *TLS* socket in default , if you want the fastest speed in development stage, you can use it without *TLS* :

```
from PyPlcnextRsc import Device, ExtraConfigure, GUISupplierExample

conf = ExtraConfigure()
conf.useTls = False
with Device('192.168.1.10', secureInfoSupplier=GUISupplierExample, port=41111,␣
↪config=conf) as device:
    ...
```

Tip: Users should also create a new TCPGateWay to make it valid :

Edit file **/etc/plcnext/device/System/RscGateway/RscGateway.settings** , add a new label in *<Gateways>..</Gateways>*.

For example:

<TcpGatewaySettings gatewayId="2" tcpPort="41111" sessionTimeout="300000" encrypted="false" />

After you getting the connected instance of *Device* , then just use it to get raw services:

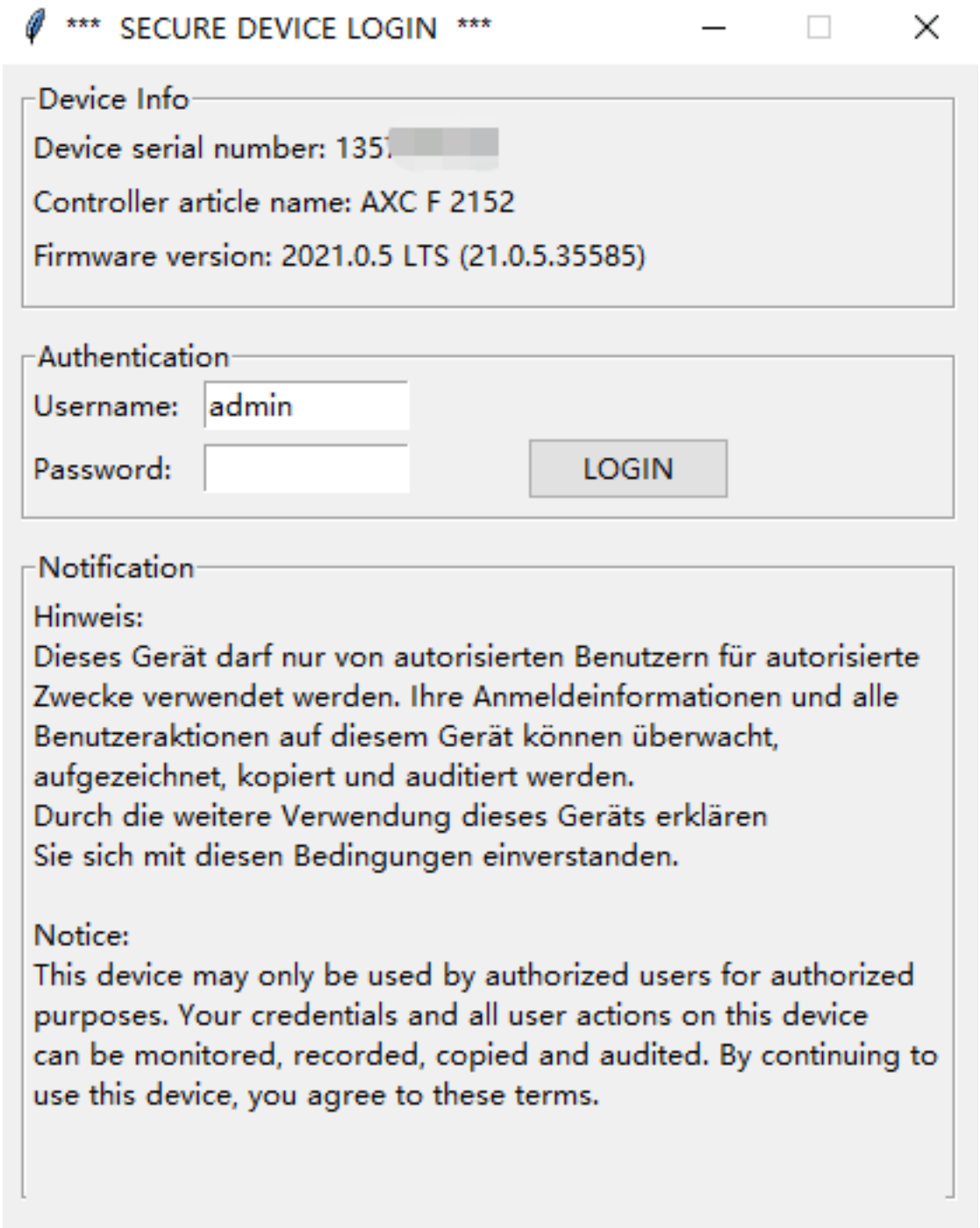
```
from PyPlcnextRsc.Arp.Plc.Gds.Services import IDataAccessService, ISubscriptionService,␣
↪IForceService
from PyPlcnextRsc.Arp.Device.Interface.Services import IDeviceInfoService

service1 = IDataAccessService(device)
service2 = ISubscriptionService(device)
service3 = IForceService(device)
service4 = IDeviceInfoService(device)
```

2.2.3 Secure Info Supplier

The *secureInfoSupplier* parameter in `Device` is a callback function that developer should supply for their application, is used to get the *username* and *password* for login. But if the user-authentication is disabled, this function will not be called, so you can leave it with None (default).

There are two demos of supplier already in the package, check the source code of them:

- *GUISupplierExample()*
- *ConsoleSupplierExample()*



```
**********************  SECURE DEVICE LOGIN  **********************
Hinweis:
Dieses Gerät darf nur von autorisierten Benutzern für autorisierte
Zwecke verwendet werden. Ihre Anmeldeinformationen und alle
Benutzeraktionen auf diesem Gerät können überwacht,
aufgezeichnet, kopiert und auditiert werden.
Durch die weitere Verwendung dieses Geräts erklären
Sie sich mit diesen Bedingungen einverstanden.

Notice:
This device may only be used by authorized users for authorized
purposes. Your credentials and all user actions on this device
can be monitored, recorded, copied and audited. By continuing to
use this device, you agree to these terms.

------------------------------------------------------------------
         Device serial number  :  1357[illegible]
      Controller article name  :  AXC F 2152
             Firmware version  :  2021.0.5 LTS (21.0.5.35585)
------------------------------------------------------------------
Username[admin]:
admin@192.168.1.10's password :
```

As you can see the source code of these example is quite simple, when the server (device) need to authentication, the function will be invoked, developer should implement their own event to get the secure info from end-user.

This function should be defined with one or zero parameter in its' signature. if one parameter is defined, then the information about the target device will be collected and passed to you as *dict*

- "General.SerialNumber" - The "SerialNumber" parameter indicates the serial number of the device.
- "General.ArticleName" - The "ArticleName" parameter indicates the device name.
- "General.Firmware.Version" - The "FirmwareVersion" parameter indicates the firmware version of the device.
- "Notification" - A message for display to users before accessing the device.
- "Address" - The ip address and port that has be configured before.

Developer could do further checking by using this, or just display to end-users. No parameter in the signature is also ok , in this case those information will not be gathered for this function.

The return of this function must be a tuple with username and password in str type

Note: Only for develop(debug) period :

Write a lambda like this is ok in develop stage, but not for release !

```
secureInfoSupplier = lambda:("admin","my_pass")
```

Warning: Always note that it is not a good habit to write the username and password directly in code, that is very dangerous.

2.2.4 Function demos

Note: The underlying type of each RscVariant returned by RSC service calls should never be assumed, but should always be checked before performing any other operation on that object.

PLC Basic

Read Status and Infos

The device interface services provide a range of functions for accessing properties of the operating system and the controller hardware. You can call the information with the following interfaces.

Link to relative service :

- *PyPlcnextRsc.Arp.Device.Interface.Services.IDeviceInfoService*
- *PyPlcnextRsc.Arp.Device.Interface.Services.IDeviceStatusService*

Tip: Get all identifiers for these services to query, visit : `# TODO`

```
from PyPlcnextRsc.Arp.Device.Interface.Services import IDeviceInfoService,
↪IDeviceStatusService
...

device_info_service = IDeviceInfoService(device)
device_status_service = IDeviceStatusService(device)
info_items = [
    "General.ArticleName",
    "General.ArticleNumber",
    "General.SerialNumber",
    "General.Firmware.Version",
    "General.Hardware.Version",
    "Interfaces.Ethernet.1.0.Mac"
]
status_items = [
    "Status.Cpu.0.Load.Percent",
    "Status.Memory.Usage.Percent",
```

(continues on next page)

(continued from previous page)

```
    "Status.ProgramMemoryIEC.Usage.Percent",
    "Status.DataMemoryIEC.Usage.Percent",
    "Status.Board.Temperature.Centigrade",
    "Status.Board.Temperature.Centigrade",
    "Status.Board.Humidity"
]
for identifier, result in zip(info_items, device_info_service.GetItems(info_items)):
    print(identifier.rjust(40) + " : " + str(result.GetValue()))
for identifier, result in zip(status_items, device_status_service.GetItems(status_
↪items)):
    print(identifier.rjust(40) + " : " + str(result.GetValue()))
```

```
>>>
                    General.ArticleName : AXC F 2152
                  General.ArticleNumber : 2404267
                   General.SerialNumber : 1357546529
                General.Firmware.Version : 2021.0.5 LTS (21.0.5.35585)
                General.Hardware.Version : 02
             Interfaces.Ethernet.1.0.Mac : 00:A0:45:A2:C0:5A
               Status.Cpu.0.Load.Percent : 5
             Status.Memory.Usage.Percent : 38
   Status.ProgramMemoryIEC.Usage.Percent : 1
      Status.DataMemoryIEC.Usage.Percent : 1
     Status.Board.Temperature.Centigrade : 52
     Status.Board.Temperature.Centigrade : 52
                  Status.Board.Humidity : 14
```

Querying notifications

Notifications that have been saved by means of the Notification Logger can be queried via RSC interfaces, and one that is already provided with the PLCnext Runtime is this INotificationLoggerService. You can use filter criteria for the query in order to specify the query. You can query all archives or the notifications of one specific archive.

For Detail Instructions, please refer to

https://www.plcnext.help/te/Service_Components/Notifications/Notification_manager.htm

A basic demo example of reading notifications:

```
from PyPlcnextRsc import Device, GUISupplierExample
from PyPlcnextRsc.Arp.Services.NotificationLogger.Services import␣
↪INotificationLoggerService, NotificationFilter, SortOrder

if __name__ == "__main__":
    with Device('192.168.1.10', secureInfoSupplier=GUISupplierExample) as device:
        notification_logger_service = INotificationLoggerService(device)

        # leave all filter field ignore
        _filter = NotificationFilter(
            StoredIdLowerLimit=0,
```

(continues on next page)

(continued from previous page)

```
            StoredIdUpperLimit=0,
            NotificationNameRegExp="",
            SenderNameRegExp="",
            TimestampBefore="",
            TimestampAfter="",
            SeverityLowerLimit="",
            SeverityUpperLimit=""
        )

        sn = notification_logger_service.QueryStoredNotifications(
            archives=["default"],
            Filter=_filter,
            limit=10,
            sortOrder=SortOrder.TimestampDesc,
            language="en"
        )

        for n in sn:
            print(f"ID:{n.Id}\t\t  Severity={n.Severity}\t\t  Sender={n.SenderName}\t\t␣
↪Payload={n.Payload}")
```

```
>>>
    ID:466                Severity=Info             Sender=System Manager               ␣
↪   Payload=('SystemManager state changed: Running, error=false, warning=false',)
    ID:465                Severity=Info             Sender=PLC Manager                  ␣
↪Payload=('Plc state changed: Stop (warm) ==> Running',)
    ID:464                Severity=Info             Sender=Device Interface             ␣
↪   Payload=('Link state changed: interface 1, port 1, status: Up',)
    ID:463                Severity=Info             Sender=System Manager               ␣
↪   Payload=('SystemManager state changed: Stop, error=false, warning=false',)
    ID:462                Severity=Info             Sender=PLC Manager                  ␣
↪Payload=('Plc state changed: Ready ==> Stop (warm)',)
    ID:461                Severity=Internal             Sender=PROFINET Controller      ␣
↪Payload=('Led state changed: Arp.Io.PnC (Controller), BF=Off, SF=Off',)
    ID:460                Severity=Info             Sender=PLC Manager                  ␣
↪Payload=('Plc state changed:  ==> Ready',)
    ID:459                Severity=Internal             Sender=PROFINET Controller      ␣
↪Payload=('Led state changed: Arp.Io.PnC (Controller), BF=On, SF=Off',)
    ID:458                Severity=Internal             Sender=PROFINET Device          ␣
↪   Payload=('Led state changed: Arp.Io.PnD (Device), BF=On, SF=Off',)
    ID:457                Severity=Info             Sender=System Manager               ␣
↪   Payload=('SystemManager state changed: Ready, error=false, warning=false',)
```

Control device

#TODO doc

Setting device

#TODO doc

Variables

Primitive

#TODO doc

Complex

Here is a helper tool for user to operate *Complex* data type, it is *DataTypeStore*

The *DataTypeStore* is mainly used to create the instance of *Complex* variable, make value in python can be equivalent to *IEC61131*.

Have a see to the basic typical example:

```
from PyPlcnextRsc import Device, RscVariant, GUISupplierExample
from PyPlcnextRsc.Arp.Plc.Gds.Services import IDataAccessService, WriteItem
from PyPlcnextRsc.tools import DataTypeStore

if __name__ == "__main__":
  TypeStore = DataTypeStore.fromString(
    """
    TYPE

    DemoStruct : STRUCT
        Field1 : INT;
        Field2 : BOOL;
    END_STRUCT

    DemoArray : ARRAY[0..10] OF INT;

    END_TYPE
    """)
  with Device('192.168.1.10', secureInfoSupplier=GUISupplierExample) as device:
    # create DemoStruct
    demo1 = TypeStore.NewSchemaInstance("DemoStruct")
    demo1.Field1 = 123
    demo1.Field2 = True
    # create DemoArray
    demo2 = TypeStore.NewSchemaInstance("DemoArray")
    demo2[:] = [i * 2 for i in range(11)]
    # get raw data access service
    data_access_service = IDataAccessService(device)
```

(continues on next page)

(continued from previous page)

```
    # Write demo1 to PLCnext
    data_access_service.Write((WriteItem("Arp.Plc.Eclr/demo1", RscVariant.of(demo1)),))
    # Read demo1
    read_item = data_access_service.Read(("Arp.Plc.Eclr/demo1",))[0]
    rcv_demo1 = TypeStore.ReceiveAsSchemaInstance("DemoStruct", read_item.Value)
    print(rcv_demo1)
    # ---------------
    data_access_service.Write((WriteItem("Arp.Plc.Eclr/demo2", RscVariant.of(demo2)),))
    read_item = data_access_service.Read(("Arp.Plc.Eclr/demo2",))[0]
    rcv_demo2 = TypeStore.ReceiveAsSchemaInstance("DemoArray", read_item.Value)
    print(rcv_demo2)
```

Let's now start with *DataTypeStore*.

There are two typical way to create it. The example showed above is directly using *str* value, another way to create us using file:

```
MyTypeStore = DataTypeStore.fromFile('MyType.txt')
```

The code for creating the *DataTypeStore* is the same to *PLCnext Engineer*, so for most case just copy the *DataType* code from *PLCnext Engineer* is ok.

Note: Known limits:

Enum and **User-defined String** is not implement yet! please use its' origin type instead.

If a type is referenced another type, the type which is referenced must be declared before, this is a little different from *PLCnext Engineer*

User default value is not supported , but will come soon.

DataTypeStore also support nested variable , so you can use any DateType in theory, such as:

```
TYPE
UnusualArray : ARRAY[-10..10] OF INT;
MyArray :ARRAY[0..10] OF INT;
MyArray2 : array [0..2] OF MyArray;
MyArray3 : ARRAY[0..1] OF MyArray2;
MyArray3plus: ARRAY[0..1] OF array [0..2] OF ARRAY[0..10] OF INT;

MyStruct0 : STRUCT
    Field1 : INT;
    Field2 : BOOL;
END_STRUCT

ArrayOfStruct : ARRAY[0..2] OF MyStruct0;

StructWithArray : STRUCT
    Field1 : ARRAY[0..10] OF INT;
    Field2 : BOOL;
END_STRUCT;

OtherStruct : STRUCT
```

(continues on next page)

(continued from previous page)

```
    Field1 : INT;
    Field2 : BOOL;
    F3:MyStruct0;
    F4:MyArray;
END_STRUCT

// Support Comment
/*
Foobar : STRUCT
    Field1 : ARRAY[0..10] OF INT;
    Field2 : BOOL;
END_STRUCT;
*/
# Also you can use pound sign to comment , this is very convenient for Python IDE to␣
↪Auto comment lines.
END_TYPE
```

Let's study with this struct:

The following code is the typical way for writing it to PLCnext

```
TypeStore = DataTypeStore.fromString(
    """
    TYPE

    St_Basic : STRUCT
        Field1 : INT;
        Field2 : BOOL;
        Field3 : DWORD;
    END_STRUCT

    St_Basic2 : STRUCT
        Field1 : St_Basic;
        Field2 : St_Basic;
    END_STRUCT

    Arr3_Str : ARRAY[0..2] OF STRING;
    Arr3_StBasic: ARRAY[0..2] OF St_Basic;
    Arr3_Time: ARRAY[0..3] OF TIME;

    MyComplexStruct : STRUCT
        xBool  : BOOL;
        xINT   : INT;
        xULINT : ULINT;
        xDWORD : DWORD;
        xStr   : STRING;
        xTIME  : TIME;
        stBasic: St_Basic;
        arrINT : ARRAY[0..2] OF INT; //Anonymous array
        arrStr : Arr3_Str;
        stStBasic2 : St_Basic2;
    END_STRUCT

    END_TYPE
    """
)
with Device('192.168.1.10', secureInfoSupplier=GUISupplierExample) as device:
    MyComplexStruct = TypeStore.NewSchemaInstance("MyComplexStruct")
    MyComplexStruct.xBool = True
    MyComplexStruct.xINT = 2152
    MyComplexStruct.xULINT = 18446744073709551615
    MyComplexStruct.xDWORD = 0xAABBCCDD  # this is just int actually
    MyComplexStruct.xStr = 'Hello World!!!'
    MyComplexStruct.xTIME = 1 * 86400000 + 2 * 3600000 + 3 * 60000 + 4 * 1000 + 567  # T
↪#1d2h3m4.567s
    MyComplexStruct.stBasic.Field1 = 123
    MyComplexStruct.stBasic.Field2 = True
    MyComplexStruct.stBasic.Field3 = 0x00112233
    MyComplexStruct.arrINT[0] = 100
    MyComplexStruct.arrINT[1] = 200
    MyComplexStruct.arrINT[2] = 300
    MyComplexStruct.arrStr[0] = "Hello"
    MyComplexStruct.arrStr[2] = "World"
    MyComplexStruct.stStBasic2.Field1.Field1 = 1000
```

(continues on next page)

(continued from previous page)

```
    MyComplexStruct.stStBasic2.Field1.Field2 = False
    MyComplexStruct.stStBasic2.Field1.Field3 = 0xABCD0123
    MyComplexStruct.stStBasic2.Field2.Field1 = 1000
    MyComplexStruct.stStBasic2.Field2.Field2 = True
    MyComplexStruct.stStBasic2.Field2.Field3 = 0x0123ABCD
    # Note : write struct or array must use **IDataAccessService.Write**
    #        the **IDataAccessService.WriteSingle** not support for complex data
    err = IDataAccessService(device).Write((WriteItem("Arp.Plc.Eclr/myComplexStruct",␣
↪RscVariant.of(MyComplexStruct)),))[0]
    if err == DataAccessError.NONE:
        print("Success")
```

For senior programmer, the following code is more friendly and high efficiency to you :

```
MyComplexStruct = TypeStore.NewSchemaInstance("MyComplexStruct")
Make_St_Basic = lambda: TypeStore.NewSchemaInstance("St_Basic")
Make_St_Basic2 = lambda: TypeStore.NewSchemaInstance("St_Basic2")

MyComplexStruct.stBasic = Make_St_Basic()(1, True, Field3=0x00112233)  # __call__ for a␣
↪struct will fill all the fields

# Direct set array data by slice
# Note:     MyComplexStruct.arrINT = [100, 200, 300] is a syntactic sugar in
#           'PyPlcnextRsc.common.objects.rsc_struct.RscStructBuilder' since V0.1.4 , see␣
↪the source code to learn more
#
#           Here the array auto created is a Subclass of python list
#           so you can use it just like list, this is the reason why you can use slice
MyComplexStruct.arrINT = [100, 200, 300]  # This is equal to MyComplexStruct.arrINT[:] =␣
↪[100, 200, 300]
MyComplexStruct.arrStr = ["Hello", "", "World"]

# since V0.1.4 this way is supported
MyComplexStruct.stStBasic2 = { # this example use dict, and also can use tuple or list
    "Field1": (1,    # use tuple or list *MUST* have the same order and size
               True,
               0x121212
               ),
    "Field2": {'Field1': 1, # using dict just fill the field you want, those fields you␣
↪ignored will remain default value
               'Field2': True,
               'Field3': 0x343434 # if this line is removed , then 'Field3' = 0x0
               }
}

# Note: it is not necessary to make your tuple(list or dict) start from root everytime,
# if it has a deep path, the following code is more friendly and it act all the same␣
↪with the above one.
MyComplexStruct.stStBasic2.Field1 = (1, True, 0x121212)
MyComplexStruct.stStBasic2.Field2 = {'Field1': 1, 'Field2': True, 'Field3': 0x343434}
```

(continues on next page)

(continued from previous page)

```
# the Original way (before V0.1.4)
# innerSt_Basic2 = Make_St_Basic2()
# innerSt_Basic2.Field1 = Make_St_Basic()(1, True, 0x121212)
# innerSt_Basic2.Field2 = Make_St_Basic()(2, False, 0x343434)
## Optional: more directly instead of above, use this:
## innerSt_Basic2(Make_St_Basic()(1, True, 0x121212), Make_St_Basic()(2, False,␣
↪0x343434))
# MyComplexStruct.stStBasic2 = innerSt_Basic2

# Note : write the struct or array must use **IDataAccessService.Write** ,
#        **IDataAccessService.WriteSingle** not support for complex data
err = IDataAccessService(device).Write((WriteItem("Arp.Plc.Eclr/myComplexStruct",␣
↪RscVariant.of(MyComplexStruct)),))[0]
if err == DataAccessError.NONE:
    print("Success")
```

Note: In the created instance , all values have been set to its' default value, means that if user don't change the element (field) in it, it can also send to PLCnext successfully , but all elements (or fields) are *0* , *0.0* , *False* or *""(empty str)*

As for read data:

```
with Device('192.168.1.10', secureInfoSupplier=GUISupplierExample) as device:
    complexStruct_ForSend = TypeStore.NewSchemaInstance("MyComplexStruct")
    Make_St_Basic = lambda: TypeStore.NewSchemaInstance("St_Basic")
    Make_St_Basic2 = lambda: TypeStore.NewSchemaInstance("St_Basic2")
    service = IDataAccessService(device)

    # Prepare value to send
    complexStruct_ForSend.stBasic = Make_St_Basic()(1, True, Field3=0x00112233)
    complexStruct_ForSend.arrINT[:] = [100, 200, 300]

    err = service.Write((WriteItem("Arp.Plc.Eclr/myComplexStruct", RscVariant.
↪of(complexStruct_ForSend)),))[0]
    if err == DataAccessError.NONE:
        print("Write Success")

    read_item = service.Read(("Arp.Plc.Eclr/myComplexStruct",))[0]
    if read_item.Error == DataAccessError.NONE:
        # Use **TypeStore.ReceiveAsSchemaInstance** to convert RscVariant object
        # to python value which is equivalence to'MyComplexStruct'
        complexStruct_Received = TypeStore.ReceiveAsSchemaInstance("MyComplexStruct",␣
↪read_item.Value)

        # you can directly compare these two objects :
        assert complexStruct_Received == complexStruct_ForSend
        complexStruct_ForSend.xINT = 6666
        assert complexStruct_Received != complexStruct_ForSend
        complexStruct_ForSend.xINT = 0
        complexStruct_ForSend.arrINT[2] = 5
        assert complexStruct_Received != complexStruct_ForSend
```

(continues on next page)

(continued from previous page)

```
        print(f"complexStruct_Received.stBasic = {complexStruct_Received.stBasic}")  #␣
↪direct print the struct
        print(f"complexStruct_Received.stBasic.Field1={complexStruct_Received.stBasic.
↪Field1}")
        print(f"complexStruct_Received.stBasic.Field2={complexStruct_Received.stBasic.
↪Field2}")
        print(f"complexStruct_Received.stBasic.Field3={complexStruct_Received.stBasic.
↪Field3}")

        # more common way is to get the reference of inner object
        inner_struct = complexStruct_Received.stBasic
        print(f"inner_struct.Field2={inner_struct.Field2}")

        print(f"complexStruct_Received.arrINT={complexStruct_Received.arrINT}")
        # Remember, the array of there is subclass instance of list
        # just treat it as list
        for idx, element in enumerate(complexStruct_Received.arrINT):
            print(f"arrINT<{idx}>{element}")
```

```
>>>
Write Success
complexStruct_Received.stBasic = stBasic(Field1=1, Field2=True, Field3=1122867)
complexStruct_Received.stBasic.Field1=1
complexStruct_Received.stBasic.Field2=True
complexStruct_Received.stBasic.Field3=1122867
inner_struct.Field2=True
complexStruct_Received.arrINT=RscList<Int16>[100, 200, 300]
arrINT<0>100
arrINT<1>200
arrINT<2>300
```

There are some advanced usages:

```
from PyPlcnextRsc import Device, GUISupplierExample, RscVariant
from PyPlcnextRsc.Arp.Plc.Gds.Services import IDataAccessService, WriteItem,␣
↪DataAccessError
from PyPlcnextRsc.tools import DataTypeStore

TypeStore = DataTypeStore.fromString(
    """
    // Anonymous 4-dimension
    CrazyArray : ARRAY[0..2] OF ARRAY[0..2] OF ARRAY[0..2] OF ARRAY[0..2] OF STRING;

    // Disassemble 4-dimension
    Inner0 : ARRAY[0..2] OF STRING;
    Inner1 : ARRAY[0..2] OF Inner0;
    Inner2 : ARRAY[0..2] OF Inner1;
    CrazyArray2 : ARRAY[0..2] OF Inner2;

    // ArrayOfStruct
    InnerStruct : STRUCT
            Field1 : INT;
```

(continues on next page)

(continued from previous page)

```
            Field2 : BOOL;
    END_STRUCT
    ArrayOfStruct : ARRAY[0..2] OF InnerStruct;

    // Chain
    StCell : STRUCT
        Field1 : String;
    END_STRUCT
    ArrCell:ARRAY[0..1] of StCell;
    StCell2:STRUCT
        Field1 : ArrCell;
    END_STRUCT
    Chain:ARRAY[0..1] of StCell2;
    """
)
with Device('192.168.1.10', secureInfoSupplier=GUISupplierExample) as device:
    service = IDataAccessService(device)
    MyCrazyArray = TypeStore.NewSchemaInstance("CrazyArray")

    # Anonymous 4-dimension
    for a in range(3):
        for b in range(3):
            for c in range(3):
                for d in range(3):
                    MyCrazyArray[a][b][c][d] = f"I am {a} {b} {c} {d}"

    err = service.Write((WriteItem("Arp.Plc.Eclr/MyCrazyArray", RscVariant.
↪of(MyCrazyArray)),))[0]
    assert err == DataAccessError.NONE

    # Disassemble 4 - dimension
    # Of course you can still use the above way, they are the same
    # this example just to show you how to generate inner array, it will be useful in␣
↪some case.

    def Inner0(a, b, c):
        ret = TypeStore.NewSchemaInstance("Inner0")
        ret[:] = [f"I am {a} {b} {c} {i}" for i in range(3)]
        return ret

    def Inner1(a, b):
        ret = TypeStore.NewSchemaInstance("Inner1")
        ret[:] = [Inner0(a, b, c) for c in range(3)]
        return ret

    def Inner2(a):
        ret = TypeStore.NewSchemaInstance("Inner2")
        ret[:] = [Inner1(a, b) for b in range(3)]
        return ret

    MyCrazyArray2 = TypeStore.NewSchemaInstance("CrazyArray2")
    MyCrazyArray2[:] = [Inner2(a) for a in range(3)]
```

(continues on next page)

(continued from previous page)

```
    err = service.Write((WriteItem("Arp.Plc.Eclr/MyCrazyArray2", RscVariant.
↪of(MyCrazyArray2)),))[0]
    assert err == DataAccessError.NONE

    # ArrayOfStruct
    MyArrayOfStruct = TypeStore.NewSchemaInstance("ArrayOfStruct")
    MyArrayOfStruct[0].Field1 = 100  # typical way
    MyArrayOfStruct[0].Field2 = True
    # Since V0.1.4 : if element is struct type in array , use this method is quick !
    MyArrayOfStruct[1] = {"Field1": 200, "Field2": False}  # or (200,False) or [200,
↪False]
    MyArrayOfStruct[2] = (300, True)
    # Since V0.1.4 : and  this can override all above values (this example showed '[:]'␣
↪you can
    # use other slice which you want such as '[0:8]' '[1:-1]'... )
    MyArrayOfStruct[:] = [{"Field1": 200, "Field2": False}, (200, True), (300, True)]

    # # before V0.1.4
    # MyArrayOfStruct[1] = TypeStore.NewSchemaInstance("InnerStruct")(Field1=200,␣
↪Field2=False)
    # # new struct instance using attribute to fill fields
    # innerStruct = TypeStore.NewSchemaInstance("InnerStruct")
    # innerStruct.Field1 = 300
    # innerStruct.Field2 = True
    # MyArrayOfStruct[2] = innerStruct
    # MyArrayOfStruct[:] = [
    #     TypeStore.NewSchemaInstance("InnerStruct")(Field1=200, Field2=False),
    #     TypeStore.NewSchemaInstance("InnerStruct")(Field1=200, Field2=False),
    #     TypeStore.NewSchemaInstance("InnerStruct")(Field1=200, Field2=False)
    # ]

    err = service.Write((WriteItem("Arp.Plc.Eclr/MyArrayOfStruct", RscVariant.
↪of(MyArrayOfStruct)),))[0]
    assert err == DataAccessError.NONE

    # Chain
    MyChain = TypeStore.NewSchemaInstance("Chain")
    MyChain[0].Field1[0].Field1 = "Hello !!"
    err = service.Write((WriteItem("Arp.Plc.Eclr/MyChain", RscVariant.of(MyChain)),))[0]
    assert err == DataAccessError.NONE
```

DataLogger

For Detail Instructions, please refer to

> https://www.plcnext.help/te/Service_Components/Remote_Service_Calls_RSC/RSC_IDataLoggerService2.htm

The DataLogger is a service component that transfers real-time data from the GDS to a database for recording and storage purposes. When starting and stopping the PLCnext Technology firmware, a configured DataLogger session is started and stopped automatically. The DataLogger then collects the task-synchronous values of the configured GDS ports with a given sampling rate and stores them with a time stamp into a database. To learn more about the DataLogger in general, read the DataLogger topic, Click.

A DataLogger instance can be configured with a configuration file, or by means of PLCnext Engineer (firmware 2020.6 or higher), or with the RSC IDataLoggerService2 as described in this topic.

Note: DataLogger sessions initiated via this RSC service will not perform the Download Changes command in PLCnext Engineer. This is true even if sessions are not currently running but are created already. The blocking is indicated by a notification (Arp.Services.DataLogger.Error, payload string: "Dynamic session detected! Download rejected!"). Unlike sessions that are started via this RSC service, sessions that are created via an XML configuration or in PLCnext Engineer are continued even after a Download All operation.

```
import datetime
import time
from PyPlcnextRsc import Device, GUISupplierExample, RscVariant, RscType
from PyPlcnextRsc.Arp.Services.DataLogger.Services import IDataLoggerService2, ErrorCode,
↪ SessionProperty, SessionPropertyName, SinkType, TriggerRpnItem, RpnItemType

if __name__ == "__main__":
    with Device('192.168.1.10', secureInfoSupplier=GUISupplierExample) as device:
        data_logger_service = IDataLoggerService2(device)  # Get DataLoggerService2
        sessionName = "MyDb"

        # Create a datalogger session
        assert data_logger_service.CreateSession("MyDb", persistent=False) == ErrorCode.
↪NONE

        # Configure the session
        propertys = [
            SessionProperty(SessionPropertyName.SamplingInterval, RscVariant.of("100ms
↪")),
            SessionProperty(SessionPropertyName.PublishInterval, RscVariant.of("200ms")),
            SessionProperty(SessionPropertyName.BufferCapacity, RscVariant(10, RscType.
↪Uint16)),
            SessionProperty(SessionPropertyName.SinkType, RscVariant.ofEnum(SinkType.
↪Database)),
            SessionProperty(SessionPropertyName.SinkProperties,
                            RscVariant.of("writeInterval=1000;dst=/opt/plcnext/test.db;
↪rollover=true;maxFiles=25")
                            ),
        ]
        assert data_logger_service.ConfigureSession(sessionName, propertys) == ErrorCode.
↪NONE
```

(continues on next page)

(continued from previous page)

```
        # Add Variables
        errors = data_logger_service.SetVariables(sessionName, ["Arp.Plc.Eclr/A", "Arp.
↪Plc.Eclr/A1.v"])
        for err in errors:
            assert err == ErrorCode.NONE

        c = [
            TriggerRpnItem(RpnItemType.Variable, RscVariant.of("Arp.Plc.Eclr/A1.v")),
            TriggerRpnItem(RpnItemType.Constant, RscVariant.of(True)),
            TriggerRpnItem(RpnItemType.Operation, RscVariant(1, RscType.Uint8)),
        ]
        # Set Trigger
        assert data_logger_service.SetTriggerCondition(sessionName, taskName="MyTask",␣
↪preCycleCount=3, postCycleCount=20, triggerCondition=c) == ErrorCode.NONE

        # Start
        assert data_logger_service.StartSession(sessionName) == ErrorCode.NONE

        time.sleep(10)

        # Read
        start = datetime.datetime(2021, 8, 27, 14, 48, 30, tzinfo=datetime.timezone.utc)
        values, err = data_logger_service.ReadVariablesData(sessionName, start, datetime.
↪datetime(2021, 8, 27, 14, 49, 00, tzinfo=datetime.timezone.utc), ["Arp.Plc.Eclr/A1.v"])

        for v in values:
            print(v)
```

Files

Read file

Link to relative service :

- *PyPlcnextRsc.Arp.System.Commons.Services.Io.IFileService*

The following snippet showed an example for reading the */opt/plcnext/logs/Output.log* file and save to local, also the *Traits* of *Length* and *Permissions* be read at the same time.

```
from PyPlcnextRsc.Arp.System.Commons.Services.Io import IFileService, Traits,␣
↪FileSystemError
...

file_service = IFileService(device)
stream, traits, error = file_service.Read(Traits.Length | Traits.Permissions, "/opt/
↪plcnext/logs/Output.log")
if error == FileSystemError.NONE:
    for trait in traits:
        print(f"Trait({trait.Trait.name}) = {trait.Value.GetValue()}")
    stream.saveToFile("log.txt")
```

```
>>>
Trait(Permissions) = 509
Trait(Length) = 1974442
```

Write file

Link to relative service :

- *PyPlcnextRsc.Arp.System.Commons.Services.Io.IFileService*
- *PyPlcnextRsc.Arp.System.Commons.Services.Io.TraitItem*
- *PyPlcnextRsc.Arp.System.Commons.Services.Io.Traits*

The following snippet send two files to PLCnext `/tmp` folder, the first file is the current python file (use variable *__file__* to represent), another file is not a real file on filesystem , just python *bytes*, and you can also use *bytearray* or *str* to wrap as a stream directly. this is convenient in some case. At mean while , these two files are written with permission `owner_all | group_all | others_all` (see *Permissions*)

```
import os
from PyPlcnextRsc import RscStream, RscVariant, RscType
from PyPlcnextRsc.Arp.System.Commons.Services.Io import IFileService, TraitItem, Traits
...

file_service = IFileService(device)
# just send this py file for example
file_path = __file__
file_name = os.path.split(file_path)[-1]
# prepare RscStream object
MyFileStream = RscStream.ofFile(file_path)
MyDataStream = RscStream.ofData(b'echo Hello World !')
# 511 = owner_all | group_all | others_all
permissionTrait = TraitItem(Traits.Permissions, RscVariant(511, RscType.Int32))
file_service.Write(f'/tmp/{file_name}', overwrite=True, traitItems=(permissionTrait,),␣
↪data=MyFileStream)
file_service.Write(f'/tmp/say_hello.sh', overwrite=True, traitItems=(permissionTrait,),␣
↪data=MyDataStream)
```

Read directory

Link to relative service :

- *PyPlcnextRsc.Arp.System.Commons.Services.Io.IDirectoryService*
- *PyPlcnextRsc.Arp.System.Commons.Services.Io.IFileService*
- *PyPlcnextRsc.Arp.System.Commons.Services.Io.Traits*

This example shows a basic case of how to download a directory from PLCnext to local filesystem, all files **(searchPattern= ” * “)** in `/opt/plcnext/projects/PCWE` will be download to `receivedPCWE` folder

Note: This is only a basic demo, users should make more exception handler in their code.

```
import os
import platform
from PyPlcnextRsc.Arp.System.Commons.Services.Io import IDirectoryService, IFileService,␣
↪Traits
...

# check if we are running on Windows
isWin = platform.system().lower() == 'windows'

# IDirectoryService for enumerating filesystem
directory_service = IDirectoryService(device)
# IFileService for downloading file
file_service = IFileService(device)

save_folder = "receivedPCWE"
remote_folder = "/opt/plcnext/projects/PCWE"

# Read all entrys of the target path on device
entrys = directory_service.EnumerateFileSystemEntries(remote_folder, searchPattern="*",␣
↪recursive=True)
for entry in entrys:
    relative_path = entry.Path.removeprefix(remote_folder + '/')
    if isWin:
        relative_path.replace('/', '\\')
    save_path = os.path.join(save_folder, relative_path)

    if entry.IsFile:
        os.makedirs(os.path.split(save_path)[0], exist_ok=True)
        file_service.Read(Traits.NONE, entry.Path)[0].saveToFile(save_path)
    else:
        os.makedirs(save_path, exist_ok=True)
```

Secure

Change password

Link to relative service :

- *PyPlcnextRsc.Arp.System.Security.Services.IPasswordConfigurationService2*

Warning: The greater the ability, the greater the responsibility

```
from PyPlcnextRsc.Arp.System.Security.Services import IPasswordConfigurationService2
password_configuration_service = IPasswordConfigurationService2(device)
password_configuration_service.changePassword("testCount", oldPassword="123456",␣
↪newPassword="654321")
# This operation is an administration function ,if you are admin , you can set password␣
↪directly:
password_configuration_service.setPassword("testCount", newPassword="654321")
```

2.3 Framework Reference

2.3.1 Device Object

class PyPlcnextRsc.common.device.**ExtraConfigure**
Additional configure for RscClient

- timeout - default is 10s
- useTls - default is True
- keepAlive_ms - default is 30000ms

class PyPlcnextRsc.common.device.**Device**(*ip: str*, *user: Optional[str] = None*, *passwd: Optional[str] = None*, *port: int = 41100*, *config: Optional[PyPlcnextRsc.common.device.ExtraConfigure] = None*, *secureInfoSupplier: Optional[Callable[[Optional[dict]], Tuple[str, str]]] = None*)
PLCnext Device Object ,this is the main object for end-user.

In order to get any RSC service , this object must be created and connect (log-in) successfully.

Warning: Always use *secureInfoSupplier* to supply security information instead of use 'user' and 'passwd' argument

Usage:

Typical way : using with-block for auto connect and dispose , and use *secureInfoSupplier* to provide secure login-information.

```
from PyPlcnextRsc import Device, GUISupplierExample

if __name__ == "__main__":
    with Device('192.168.1.10', secureInfoSupplier=GUISupplierExample) as␣
↪device:
        ...
```

Parameters

- **ip** (*str*) – the IP address of the target PLCnext for connecting
- **user** (*str*) – user name for login if user authentication is enabled. (deprecated ,use secureInfoSupplier instead)
- **port** (*int*) – socket port of target , default is set to *41100*
- **passwd** (*str*) – passwd for login if user authentication is enabled. (deprecated ,use secureInfoSupplier instead)
- **config** (ExtraConfigure) – additional setting for this client , such as switch of *TLS* and socket time-out

- **secureInfoSupplier** (*Callable[[Optional[dict]],Tuple[str,str]]*) – a callback function for getting secure info. the function's signature can have no or only one argument, if one argument is in signature,then during login period the details about target (such as serial number ,firmware-version) will be passed by the argument

---

Note: The function will not be called if **user-authentication** is *disabled* on target PLCnext

---

Raises #TODO

PyPlcnextRsc.common.device.**GUISupplierExample**(*detail*)

A simple GUI for authentication login based on tkinter

Note: This GUI will not be called if user-authentication is disabled.

PyPlcnextRsc.common.device.**ConsoleSupplierExample**(*detail*)

A simple console interface for authentication login based on getpass module

---

Note: This will not be called if user-authentication is disabled.

If is running by PyCharm , the `getpass module`'s behavior is changed, the password will be echoed. This will not happen if you run this directly through python

---

2.3.2 Type Tags

class PyPlcnextRsc.common.tag_type.**RscType**(*value*)

Datatypes supported by Rsc.

Null = 0

No data type set

End = 255

End of stream (EOS)

Void = 1

void or null object

Bool = 2

bool type

Char = 3

16 bit character

Int8 = 4

signed 8 bit integer (I1)

Uint8 = 5

unsigned 8 bit integer (U1)

Int16 = 6

signed 16 bit integer (I2)

Uint16 = 7

unsigned 16 bit integer (U2)

Int32 = 8
signed 32 bit integer (I4)

Uint32 = 9
unsigned 32 bit integer (U4)

Int64 = 10
signed 64 bit integer (I8)

Uint64 = 11
unsigned 64 bit integer (U8)

Real32 = 12
32 bit floating point number (R4)

Real64 = 13
64 bit floating point number (R8)

Struct = 18
Complex datatype

Utf8String = 19
Utf-8 string

String = 14
String with undefined format. Deprecated with remoting version 4 used by common.In common context with at least remoting version 4 RscType String is mapped to Utf8String

Array = 20
Array type

Datetime = 23
Datetime

Version = 24
Version

Guid = 25
Universal unique ID

AnsiString = 26
Ansi string, not implemented in common context

Object = 28
Object type handled by common as *RscVariant*

Utf16String = 30
Utf-16 string, not implemented in common context

Stream = 34
Stream type to marshal large data packets

Enumerator = 35
Enumerator type

SecureString = 36
String for security context

Enum = 37
Enum type

Dictionary = 38
Dictionary type

SecurityToken = 39
Security token needed for security Services

Exception = 40
Exception

IecTime = 41
IEC type: TIME [int32]

IecTime64 = 42
IEC type: LTIME [int64]

IecDate = 43
IEC type: DATE [N/A]

IecDate64 = 44
IEC type: LDATE [int64]

IecDateTime = 45
IEC type: DATE_AND_TIME, DT [N/A]

IecDateTime64 = 46
IEC type: LDATE_AND_TIME, LDT [int64]

IecTimeOfDay = 47
IEC type: TIME_OF_DAY, TOD [N/A]

IecTimeOfDay64 = 48
IEC type: LTIME_OF_DAY, LTOD [int64]

class PyPlcnextRsc.common.tag_type.**IecType**
Concrete annotation for *IEC61131* data space

this is just a helper map from *IEC61131* data type to *RscType*

Null = 0
Mapped to *PyPlcnextRsc.common.tag_type.RscType.Null*

TIME = 41
Mapped to *PyPlcnextRsc.common.tag_type.RscType.IecTime*

LTIME = 42
Mapped to *PyPlcnextRsc.common.tag_type.RscType.IecTime64*

LDATE = 44
Mapped to *PyPlcnextRsc.common.tag_type.RscType.IecDate64*

LDATE_AND_TIME = 46
Mapped to *PyPlcnextRsc.common.tag_type.RscType.IecDateTime64*

LTIME_OF_DAY = 48
Mapped to *PyPlcnextRsc.common.tag_type.RscType.IecTimeOfDay64*

BOOL = 2
Mapped to *PyPlcnextRsc.common.tag_type.RscType.Bool*

STRING = 19
Mapped to *PyPlcnextRsc.common.tag_type.RscType.Utf8String*

LREAL = 13
Mapped to *PyPlcnextRsc.common.tag_type.RscType.Real64*

REAL = 12
Mapped to *PyPlcnextRsc.common.tag_type.RscType.Real32*

LWORD = 11
Mapped to *PyPlcnextRsc.common.tag_type.RscType.Uint64*

DWORD = 9
Mapped to *PyPlcnextRsc.common.tag_type.RscType.Uint32*

WORD = 7
Mapped to *PyPlcnextRsc.common.tag_type.RscType.Uint16*

BYTE = 5
Mapped to *PyPlcnextRsc.common.tag_type.RscType.Uint8*

LINT = 10
Mapped to *PyPlcnextRsc.common.tag_type.RscType.Int64*

DINT = 8
Mapped to *PyPlcnextRsc.common.tag_type.RscType.Int32*

INT = 6
Mapped to *PyPlcnextRsc.common.tag_type.RscType.Int16*

SINT = 4
Mapped to *PyPlcnextRsc.common.tag_type.RscType.Int8*

ULINT = 11
Mapped to *PyPlcnextRsc.common.tag_type.RscType.Uint64*

UDINT = 9
Mapped to *PyPlcnextRsc.common.tag_type.RscType.Uint32*

UINT = 7
Mapped to *PyPlcnextRsc.common.tag_type.RscType.Uint16*

USINT = 5
Mapped to *PyPlcnextRsc.common.tag_type.RscType.Uint8*

2.3.3 Tools

DataTypeStore

PyPlcnextRsc.tools.PlcDataTypeSchema.**NewSchemaInstance**(*schema*)
Construct a new data type instance defined by schema this instance is used for Write complex data to PLCnext

---

Note: In the created instance , all values have been set to its' default value, means that if user don't change the element (field) in it, it can also send to PLCnext successfully , but all elements (or fields) are *0* , *0.0* , *False* or *""(empty str)*

Parameters **schema** – type schema, get from *PyPlcnextRsc.tools.PlcDataTypeSchema.DataTypeStore.__getitem__()*

Returns a new data type instance for user to fill elements or fields

PyPlcnextRsc.tools.PlcDataTypeSchema.**ReceiveAsSchemaInstance**(*schema*, *variant:* PyPlcnextRsc.common.objects.rsc_variant.RscVariant, *builderMode: bool = False*)
Receive the certain value from *RscVariant* as data_type defined in schema

Parameters

- **schema** – type schema
- **variant** (*RscVariant*) – from IDataAccessService or ISubscriptionService or some other service , which represent the meta-value of the data_type defined in schema
- **builderMode** (*bool*) – if set true, all the struct received will be the *RscStructBuilder*, so it is possible for user to change the fields' value and send back to device directly.

Returns final value , which has the same element type or fields defined in schema

class PyPlcnextRsc.tools.PlcDataTypeSchema.**DataTypeStore**(*SIO*)

This is a helper function for python user to create the equivalent variable model to *IEC61131*, it is the most convenient way to construct a complex value such as **Array** and **Struct** for sending or receiving.

Basic usage:

```
from PyPlcnextRsc import Device, RscVariant, GUISupplierExample
from PyPlcnextRsc.Arp.Plc.Gds.Services import IDataAccessService, WriteItem
from PyPlcnextRsc.tools import DataTypeStore

if __name__ == "__main__":
    TypeStore = DataTypeStore.fromString(
        '''
        TYPE
        DemoStruct : STRUCT
            Field1 : INT;
            Field2 : BOOL;
        END_STRUCT
        DemoArray : ARRAY[0..10] OF INT;
        END_TYPE
        ''')
    with Device('192.168.1.10', secureInfoSupplier=GUISupplierExample) as␣
↪device:
        # create DemoStruct
        demo1 = TypeStore.NewSchemaInstance("DemoStruct")
        demo1.Field1 = 123
        demo1.Field2 = True
        # create DemoArray
        demo2 = TypeStore.NewSchemaInstance("DemoArray")
        demo2[:] = [i * 2 for i in range(11)]
        # get raw data access service
        data_access_service = IDataAccessService(device)
        # Write demo1 to PLCnext
        data_access_service.Write((WriteItem("Arp.Plc.Eclr/demo1",␣
↪RscVariant.of(demo1)),))
        # Read demo1
        read_item = data_access_service.Read(("Arp.Plc.Eclr/demo1",))[0]
        rcv_demo1 = TypeStore.ReceiveAsSchemaInstance("DemoStruct", read_
↪item.Value)
        print(rcv_demo1)
        # ---------------
        data_access_service.Write((WriteItem("Arp.Plc.Eclr/demo2",␣
↪RscVariant.of(demo2)),))
        read_item = data_access_service.Read(("Arp.Plc.Eclr/demo2",))[0]
        rcv_demo2 = TypeStore.ReceiveAsSchemaInstance("DemoArray", read_
↪item.Value)
```

(continues on next page)

(continued from previous page)

```
        print(rcv_demo2)
```

classmethod fromFile(*file_or_filename: Union[str, TextIO]*)
Create the DataTypeStore using local file.

Parameters **file_or_filename** (*TextIO or str*) – file object or the path of the local file.

classmethod fromString(*string: str*)
Using DataType string to create the DataTypeStore

Parameters **string** (*str*) – the DataType code.

__getitem__(*item*) → any
Get type schema

Parameters **item** (*str*) – type named defined in DataTypeStore

Returns type schema

NewSchemaInstance(*data_type_name: str*)
Construct a new data type instance defined by schema this instance is used for Write complex data to PLCnext

---

Note: In the created instance , all values have been set to its' default value, means that if user don't change the element (field) in it, it can also send to PLCnext successfully ,but all elements (or fields) are *0*,*0.0*,*False* or *""(empty str)*

Parameters **data_type_name** (*str*) – type named defined in DataTypeStore

Returns a new data type instance for user to fill elements or fields

ReceiveAsSchemaInstance(*data_type_name: str*, *variant:* PyPlcnextRsc.common.objects.rsc_variant.RscVariant, *builderMode: bool = False*)
Receive the certain value from PLCnext as data_type defined in schema

Parameters

- **data_type_name** (*str*) – type named defined in DataTypeStore
- **variant** (*RscVariant*) – from IDataAccessService or ISubscriptionService or some other service , which represent the meta-value of the data_type defined in schema
- **builderMode** (*bool*) – if set true, all the struct received will be the *RscStructBuilder*, so it is possible for user to change the fields' value and send back to device directly.

Returns final value , which has the same element type or fields defined in schema

2.3.4 Data Objects

General Object

RscVariant

class PyPlcnextRsc.common.objects.rsc_variant.**RscVariant**(*value*, *rscType:* PyPlcnextRsc.common.tag_type.RscType)

This is used to represent an data-object (py-value with its' RscType)

Create a RscVariant object by providing both the definite *RscType* and python value

Parameters

- **value** – Python value which to be wrapped in.
- **rscType** (*RscType*) – the definite type of the value.

Note: If you are just make RscVariant object mapped to *IEC61131*, use *IecType* is more concrete

---

classmethod **of**(*value*)

Create RscVariant from some special python object that *RscType* is clearly to tell from

> **Warning:** This is not suitable for some value that is ambiguous , for example you give number 100 use this method, but for this method it is not possible to know which *INT* (or in other word, which *RscType*)you are talking about: it shell be *PyPlcnextRsc.common.tag_type.RscType.Uint8* ? *PyPlcnextRsc.common.tag_type.RscType.Int16* ? or *PyPlcnextRsc.common.tag_type.RscType.Int64* ...?
>
> So in this case, you must use `__init__()` to give the *RscType* explicitly.

- bool value : the *PyPlcnextRsc.common.tag_type.RscType.Bool* will be filled
- str value : the *PyPlcnextRsc.common.tag_type.RscType.Utf8String* will be filled
- *RscTuple* or *RscList* : the *PyPlcnextRsc.common.tag_type.RscType.Array* will be filled
- *RscStructMeta* or *RscStructBuilder* :the *PyPlcnextRsc.common.tag_type.RscType.Struct* will be filled

Returns RscVariant instance

GetValue() → any

Get the python value in this RscVariant object.

Returns

any python value that represent the corresponding value from PLC

Note: Special case of *RscType*

- *PyPlcnextRsc.common.tag_type.RscType.Array* : the value type might be *RscList*
- *PyPlcnextRsc.common.tag_type.RscType.Struct*: the value type might be *RscStructMeta*

GetType() → *PyPlcnextRsc.common.tag_type.RscType*
Get the *RscType* corresponding to the contained value.

Return type *RscType*

GetArrayElementCtx()
This method is mainly for internal use, to get the element context while this object contains *Array*

GetFieldCount()
This method is mainly for internal use, to get the field counts while this object contain *Struct*

RscStream

class PyPlcnextRsc.common.objects.rsc_stream.**RscStream**(*bytesBuffer*)
This is used to represent a stream object

Normally used to transfer files with device

classmethod ofFile(*filePath: str*)
create a RscStream from local file directly

Parameters **filePath** (*str*) – file path of the local file

classmethod ofData(*data: Union[str, bytes, bytearray]*)
create a RscStream from data directly

Parameters **data** (*str,bytes,bytearray*) – data to be wrapped in stream object

getBufferIO() → _io.BytesIO
get the inner IO reference

Returns bytesIO in this stream

Return type BytesIO

getValue() → bytes
get all bytes in this stream

Returns data

Return type bytes

saveToFile(*filePath: str*)
save the current stream to file

Note: this method only call **open(filePath, "wb")** internal , so you must create directory before execute this method if necessary.

Parameters **filePath** (*str*) – path of the target file to Write

getSize()
get the current byte size in this stream object

Returns size of bytes

Return type int

Struct

class PyPlcnextRsc.common.objects.rsc_struct.**RscStructBuilder**(*prototype*, *defaults=None*)
This is a wrapper around RscStruct , supporting filling fields step by step

Normally this is auto created by *PyPlcnextRsc.tools.PlcDataTypeSchema*

Note: This class has overload the __eq__ method , so this builder instance is compareable to *RscStruct*

Tip: Since V0.1.4 :

'_friendlyMode':

A syntactic sugar for Array (as a field in Struct) operation has been added 'Struct.ArrField = [100, 200, 300]' in now allowed (the old way is 'Struct.ArrField[:] = [100, 200, 300]').

Usage:

```
class ST_Prototype(RscStruct):
    F0: IecAnnotation.INT
    F1: IecAnnotation.INT
    F2: IecAnnotation.BOOL


# Not possible to init fields' value separately:
# st = ST_Prototype()
# st.F0 = 0
# st.F1 = 100 # Illegal ! , because 'RscStruct' is tuple in fact
# st.F2 = False

# You can only instance it by the following way:
# st = ST_Prototype(0,100,False)

# use RscStructBuilder:
builder = RscStructBuilder(ST_Prototype)
builder(F0 = 0,F1 = 100,F2 = False) # by __call__
builder.F1 = 200    # by set attribute
builder.F2 = True

# not necessary for user to call _getRscStruct because it will auto invoke␣
↪while sending to PLC
st = builder._getRscStruct()
```

Parameters

- **prototype** (*RscStruct*) – RscStruct prototype
- **defaults** (*dict*) – dict with field name and it's default value if supplied, default it None.

class PyPlcnextRsc.common.objects.rsc_struct.**RscStructMeta**(*iterable=()*, */*)
Meta-data to represent Struct for transferring with device, it carriers all necessary information of each fields such as *RscType*

Should always used as a value in *RscVariant* , and every element are always *RscVariant* too

classmethod fromInstance(*struct_instance*)
Create RscStructMeta from a *RscStruct* instance or *RscStructBuilder*

Parameters **struct_instance** (*RscStruct* instance or *RscStructBuilder*) – struct instance, which can be an instance of *RscStruct* directly or *RscStructBuilder*

GetAsRscStruct(*struct_type*, *strictMode=True*) → NamedTuple
Generate certain RscStruct instances from provided *RscStruct* type.

Parameters

- **struct_type** (*RscStruct* type) – the type of *RscStruct*.
- **strictMode** (*bool*) – check the shape and type in meta before generate, default is *True*.

Returns the instance of *RscStruct* with all fields filled.

Return type *RscStruct*

PyPlcnextRsc.common.objects.rsc_struct.**RscStruct**
alias of NamedTuple

PyPlcnextRsc.common.objects.rsc_struct.**isRscStructInstance**(*value*)
Test whether the given value is the instance of RscStruct

PyPlcnextRsc.common.objects.rsc_struct.**isRscStructType**(*value*)
Test whether the given value is the prototype (or called 'type') of RscStruct

Array

class PyPlcnextRsc.common.objects.rsc_sequence.**RscSequence**
Bases: object

COMMON ABSTRACT CLASS FOR *RscList* & *RscTuple*

ALSO USED IN USER STRUCT DEF

setElementRscType(*rscType:* PyPlcnextRsc.common.tag_type.RscType)
Configure the sequence by passing in primitive type (*RscType*)

> **Warning:** This method only support primitive types,
>
> If next-dimension is *Array*, use *setNextDimension()* instead.
>
> else if element is *Struct*, use *setElementAnnotate()* instead.

Parameters **rscType** (*RscType*) – the element type

Returns self

setElementAnnotate(*annotate*)
Configure by annotate

Used if element type is Primitive type or *RscStruct*

Parameters **annotate** – field type annotation ,

such as 'RscTpAnnotate[int, Marshal(rscType=RscType.Int16)]' or defined *RscStruct*

Returns self

setNextDimension(*nextDimension*)
Configure by next *RscSequence* or *RscList*

> **Warning:** Only used for define a multi-dimension array.

Parameters **nextDimension** (*RscSequence* or factory of *RscList*) – next configured *RscSequence* or *RscList*

Returns self

getElementRscType() → *PyPlcnextRsc.common.tag_type.RscType*
Get the *RscType* of the element.

Returns the *RscType* of the element.

Return type *RscType*

setElementContext(*context*)
Set the DataTagContext of the element, this is used internally.

Parameters **context** – DataTagContext of the element

getElementContext()
Get the DataTagContext of the element,, this is used internally.

Returns DataTagContext

setDesireLength(*length*)
if the desire length is set, the framework will check the length before sending to device. this will auto set by *PyPlcnextRsc.tools.PlcDataTypeSchema* internally.

Parameters **length** (*int*) – the desired length of this sequence.

getDesireLength()
Get the desired length internal. if not set return -1

Returns the desired length , -1 if not set

Return type int

class PyPlcnextRsc.common.objects.rsc_sequence.**RscList**(*iterable=()*, */*)
Bases: list, *PyPlcnextRsc.common.objects.rsc_sequence.RscSequence*

Use List to represent an array for transferring with device.

Tip: Since V0.1.4 :

'_friendlyMode' :

if an element is struct type in this array , use this method is quick !

```
MyArrayOfStruct[1] = {"Field1": 200, "Field2": False}  # or (200,
↪False) or [200,False]
MyArrayOfStruct[2] = (300, True)
```

and this can override all above values (this example showed '[:]' you can use other slice which you want such as '[0:8]' '[1:-1]'…)

```
MyArrayOfStruct[:] = [{"Field1": 200, "Field2": False}, (200, True),␣
↪(300, True)]
```

classmethod factory(*funcName*, **args*)

Make a *RscList* factory, this is only a helper function to create *RscList* of same structure.

Usage:

```
middle_layer_factory = RscList.factory('createNextDimension', RscType.
↪Int16, 3)
middle1 = middle_layer_factory.create()
middle1[0].extend((1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 11))
middle1[1].extend((1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 11))
middle1[2].extend((1, 3, 2, 1, 1, 1, 1, 1, 1, 1, 11))

middle2 = middle_layer_factory.create()
middle2[0].extend((2, 1, 0, 1, 1, 1, 1, 1, 1, 1, 11))
middle2[1].extend((2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 11))
middle2[2].extend((2, 3, 2, 1, 1, 1, 1, 1, 1, 1, 11))

outer = RscList((middle1, middle2)).setNextDimension(middle_layer_
↪factory)
```

Parameters

- **funcName** (*str*) – the exist function name for configure the *RscList*.
- **args** – the function arguments corresponding to the 'funcName'.

Returns return a factory instance , which has a **create()** method to call for construct a new *RscList* by the provided parameters.

setOffset(*offset: int*)

Set the start offset of this array object , because in *IEC61131*, an array can be defined with lower bound not equal to *0*

> **Warning:** Although lower bound not equal to *0* is acceptable by this method, but still not recommend to use that kind of array definition! because it can be confused. and slice function of python can not work normally if use negative index.

Parameters **offset** (*int*) – lower bound of the array

createNextDimension(*elementRscType: PyPlcnextRsc.common.tag_type.RscType*, *count: int*, *reserve: int = 0*, *default: Optional[any] = None*)

Helper function to create next dimension ,only support for primitive python type as next dimension's element type!

Parameters

- **elementRscType** (*RscType*) – *RscType* of the element
- **count** (*int*) – the count of element in current dimension to reserve
- **reserve** – the count of element in next dimension to reserve
- **default** – the value to fill in next dimension

Returns self

reserve(*reserve: int = 0*, *default: Optional[any] = None*, *use_deepcopy: bool = False*)
Use the provided value(default) to fill the array

Parameters

- **reserve** (*int*) – element count to reserve
- **default** – the value to fill
- **use_deepcopy** (*bool*) – true if use deepcopy to fill the element,default is False

setElementRscType(*rscType:* PyPlcnextRsc.common.tag_type.RscType, *reserve: int = 0*, *default: Optional[any] = None*)
Overload the *setElementRscType()*, add the *reserve()* method.

Parameters

- **rscType** (*RscType*) – the element type
- **reserve** (*int*) – element count to reserve
- **default** – the value to fill

setElementAnnotate(*annotate*, *reserve: int = 0*, *default: Optional[any] = None*)
Overload the *setElementAnnotate()*, add the *reserve()* method.

Parameters

- **annotate** – field type annotation ,

  such as 'RscTpAnnotate[int, Marshal(rscType=RscType.Int16)]' or defined *RscStruct*
- **reserve** (*int*) – element count to reserve
- **default** – the value to fill

setNextDimension(*nextDimension*, *count: Optional[int] = None*)
Overload the *setNextDimension()*, add support for reserve.

Parameters

- **nextDimension** (*RscSequence* or factory of *RscList*) – next configured *RscSequence* or *RscList*
- **count** (*int*) – element count to reserve

Returns self

__eq__(*other*)
Overload the list.__eq__, now must the other element's *RscType* same with self's element type is possible to return True

Parameters **other** – other instance to compare

Returns true if 'other' is same with self

class PyPlcnextRsc.common.objects.rsc_sequence.**RscTuple**(*iterable=()*, */*)
Bases: tuple, *PyPlcnextRsc.common.objects.rsc_sequence.RscSequence*

Use Tuple to represent an array for transferring with device

setOffset(*offset*)
see doc from *PyPlcnextRsc.common.objects.rsc_sequence.RscList.setOffset()*

__eq__(*other*)
Overload the tuple.__eq__, now must the other element's *RscType* same with self's element type is possible to return True

Parameters `other` – other instance to compare

Returns true if 'other' is same with self

2.4 Service Reference

2.4.1 Arp.Services.NotificationLogger.Services

`class PyPlcnextRsc.Arp.Services.NotificationLogger.Services.SortOrder(value)`
Definition how to sort the queried notifications

`NONE = 0`
Do not sort

`TimestampAsc = 1`
sort by timestamp ascending

`TimestampDesc = 2`
sort by timestamp descending

`class PyPlcnextRsc.Arp.Services.NotificationLogger.Services.Severity(value)`
Enumeration of Severities for notifications

`class PyPlcnextRsc.Arp.Services.NotificationLogger.Services.NotificationFilter(StoredIdLowerLimit: int, StoredIdUpperLimit: int, NotificationNameRegExp: str, SenderNameRegExp: str, TimestampBefore: str, TimestampAfter: str, SeverityLowerLimit: str, SeverityUpperLimit: str)`
Filter specification to match notification on query or delete

`StoredIdLowerLimit: int`
Minimum matching Id (>= 0)

ower limit of the StoredId (> = 1), is ignored if = 0 In the Notification Logger, a notification is clearly identified by a StoredId (uint64). The StoredId is assigned by the Notification Logger when adding the notification to the input buffer.

`StoredIdUpperLimit: int`
Maximum matching Id (<= 1^64)

Upper limit of the StoredId (> = 1, < = 18446744073709551615, max. uint64), is ignored if = 0

NotificationNameRegExp: str
Regular expression for the notification name. Is ignored if field is empty.

SenderNameRegExp: str
Regular expression for the sender name. Is ignored if field is empty.

TimestampBefore: str
Matches all timestamps before this timestamp ,Format: YYYY-MM-ddTHH:mm:ss.SSS,Ignored if empty

TimestampAfter: str
Matches all timestamps after this timestamp ,Format: YYYY-MM-ddTHH:mm:ss.SSS,Ignored if empty

SeverityLowerLimit: str
Minimum matching Severity ,Ignored if empty

SeverityUpperLimit: str
Maximum matching Severity,Ignored if empty

class PyPlcnextRsc.Arp.Services.NotificationLogger.Services.**StoredNotification**(*Id: int*, *Archive: str*, *NotificationName: str*, *SenderName: str*, *TimeStamp: str*, *Severity: str*, *Payload: Sequence[str]*, *PayloadXml: Sequence[str]*)

Data structure for notifications from the NotificationLogger

Id: int
Id of the notification

Archive: str
name of the archive the notification was retreived from. If the same notification was stored in multiple archives this field contains a comma separated list of the archives

NotificationName: str
Name of the notification

SenderName: str
Name of the sender

TimeStamp: str
timestamp when the notification was sent

Severity: str
Severity

Payload: Sequence[str]
Formatted payload

PayloadXml: Sequence[str]
Payload as XML

class PyPlcnextRsc.Arp.Services.NotificationLogger.Services.**Notification**(*Id: int*, *NotificationNameId: int*, *Timestamp: datetime.datetime*, *Severity:* PyPlcnextRsc.Arp.Services.NotificationLogger.S *PayloadTypeId: int*, *Payload: Sequence[*PyPlcnextRsc.common.objects.

Contains meta data and paylod of a Notification

Id: int
Returns the id

NotificationNameId: int
Returns the NotificationNameId

Timestamp: datetime.datetime
Returns the timestamp

Severity: *PyPlcnextRsc.Arp.Services.NotificationLogger.Services.Severity*
Returns the Severity

PayloadTypeId: int
Returns the PayloadTypeId

Payload: Sequence[*PyPlcnextRsc.common.objects.rsc_variant.RscVariant***]**
Returns a reference to the raw payload

class PyPlcnextRsc.Arp.Services.NotificationLogger.Services.**INotificationLoggerService**
The NotificationLogger stores Notifications and provides an interface to retrieve them.

QueryStoredNotifications(*archives: Sequence[str]*, *Filter:* PyPlcnextRsc.Arp.Services.NotificationLogger.Services.NotificationFilter, *limit: int*, *sortOrder:* PyPlcnextRsc.Arp.Services.NotificationLogger.Services.SortOrder, *language: str*) → Tuple[*PyPlcnextRsc.Arp.Services.NotificationLogger.Services.StoredNotification*]

Queries notifications matching the supplied filter from the mentioned archives and returns them as StoredNotification objects

Parameters

- **archives** (*Sequence[str(max=512)]*) – List of archives to query. Empty list queries all.
- **Filter** (NotificationFilter) – filter specifications
- **limit** (*Int32*) – maximum number of returned notifications
- **sortOrder** (SortOrder) – sorting to apply
- **language** (*str(max=512)*) – translate notification payloads

Returns collection of notifications

Return type tuple[*StoredNotification*]

QueryNotifications(*archives: Sequence[str]*, *Filter:* PyPlcnextRsc.Arp.Services.NotificationLogger.Services.NotificationFilter, *limit: int*, *sortOrder:* PyPlcnextRsc.Arp.Services.NotificationLogger.Services.SortOrder, *language: str*) → Tuple[*PyPlcnextRsc.Arp.Services.NotificationLogger.Services.Notification*]

Queries notifications matching the supplied filter from the mentioned archives and returns them as Notification objects

Parameters

- **archives** (*Sequence[str(max=512)]*) – List of archives to query. Empty list queries all.
- **Filter** (NotificationFilter) – filter specifications
- **limit** (*Int32*) – maximum number of returned notifications
- **sortOrder** (SortOrder) – sorting to apply
- **language** (*str(max=512)*) – translate notification payloads

Returns collection of notifications

Return type tuple[*Notification*]

DeleteNotifications(*archives: Sequence[str]*, *Filter:* PyPlcnextRsc.Arp.Services.NotificationLogger.Services.NotificationFilter) → int

Remove notifications mathing the filter from the given archives

Parameters

- **archives** (*Sequence[str(max=512)]*) – List of archives to delete notifications from. Empty list deletes from all.
- **Filter** (NotificationFilter) – filter specification, matching notifications are removed

Returns number of deleted notifications

Return type Int32

ListArchives() → Tuple[str]

Queries a list of archives

Returns list of known archives

Return type tuple[str]

GetArchiveConfiguration(*archive: str*) → Tuple[str]

Query the configuration as XML for the given archive

> **Warning:** The operation 'GetArchiveConfiguration' is not implemented yet

Parameters **archive** (*str(max=512)*) – name of the archive

Returns XML of the configuration

Return type tuple[str]

SetArchiveConfiguration(*archive: str*, *xmlConfiguration: Sequence[str]*) → bool

Set the configuration of the given archive

> **Warning:** The operation 'SetArchiveConfiguration' is not implemented yet

Parameters

- **archive** (*str(max=512)*) – name of the archive
- **xmlConfiguration** (*Sequence[str(max=512)]*) – XML containing the configuration

Returns true on success

Return type bool

ResetArchiveConfigurationToFiles(*archive: str*) → bool

Resets the configuration of the given archive to the configuration files. All changes made by RSC are reverted.

> **Warning:** The operation 'ResetArchiveConfigurationToFiles' is not implemented yet

Parameters **archive** (*str(max=512)*) – name of the archive

Returns true on success

Return type bool

2.4.2 Arp.Services.DataLogger.Services

class PyPlcnextRsc.Arp.Services.DataLogger.Services.**DataType**(*value*)

NONE = 0
Unspecified.

Void = 1
Void - Arp C++ empty type

Bit = 2
Bit - Arp C++ data type (1 Byte)

Boolean = 3
Boolean - Arp C++ data type (1 Byte)

UInt8 = 4
UInt8 - Arp C++ data type (1 Byte)

Int8 = 5
Int8 - Arp C++ data type (1 Byte)

Char8 = 6
Char8 - Arp C++ data type (1 Byte)

Char16 = 7
Char16 - Arp C++ data type (2 Byte)

UInt16 = 8
UInt16 - Arp C++ data type (2 Byte)

Int16 = 9
Int16 - Arp C++ data type (2 Byte)

UInt32 = 10
UInt32 - Arp C++ data type (4 Byte)

Int32 = 11
Int32 - Arp C++ data type (4 Byte)

UInt64 = 12
UInt64 - Arp C++ data type (8 Byte)

Int64 = 13
Int64 - Arp C++ data type (8 Byte)

Float32 = 14
Float32 - Arp C++ data type (4 Byte)

Float64 = 15
Float64 - Arp C++ data type (8 Byte)

Primitive = 32
Limit of primitive types

DateTime = 33
C++ DateTime type

IecTime = 34
IEC type: TIME [int32]

IecTime64 = 35
IEC type: LTIME [int64]

IecDate = 36
IEC type: DATE [N/A],Not supported by PCWE.

IecDate64 = 37
IEC type: LDATE [int64]

IecDateTime = 38
IEC type: DATE_AND_TIME, DT [N/A],Not supported by PCWE.

IecDateTime64 = 39
IEC type: LDATE_AND_TIME, LDT [int64]

IecTimeOfDay = 40
IEC type: TIME_OF_DAY, TOD [N/A],Not supported by PCWE.

IecTimeOfDay64 = 41
IEC type: LTIME_OF_DAY, LTOD [int64]

StaticString = 42
Static String type

IecString = 43
Iec String type, only for internal use

ClrString = 44
.NET/C# String type, only for internal use

String = 45
C++ String type, only for internal use

Elementary = 64
Limit of elementary types.

ArrayElement = 65
ArrayOfArray

Struct = 66
Struct

Class = 67
Class

FunctionBlock = 68
Function Block

Subsystem = 69
Subsystem

Program = 70
Program

Component = 71
Component

Library = 72
Library

Complex = 254
Limit of complex types

Pointer = 512
Determines a pointer type.Pointer are declared as *PyPlcnextRsc.Arp.Plc.Commons.DataType.Elementary* kind.

Array = 1024
Determines an array type.Arrays are declared as *PyPlcnextRsc.Arp.Plc.Commons.DataType.Elementary* kind.

Enum = 2048
Determines an Enumeration type.Enums are declared as *PyPlcnextRsc.Arp.Plc.Commons.DataType.Elementary* kind.

Reference = 4096
Determines a C# reference type.Reference are declared as *PyPlcnextRsc.Arp.Plc.Commons.DataType.Elementary* kind.

BaseTypeMask = 255
For removing all flags

class PyPlcnextRsc.Arp.Services.DataLogger.Services.**ErrorCode**(*value*)
Possible error codes for data-logger rsc services.

NONE = 0
Function call succeeded.

NoSuchSession = 1
The specified session does not exist.

SessionRunning = 2
The specified session is in running state.

NoSuchVariable = 3
The specified variable does not exists.

AlreadyExists = 4
A session with the same name already exists.

OutOfMemory = 5
An attempt to allocate memory failed.

NotSupported = 6
Logging of variable not supported.

NoData = 7
No data exists for the requested time range

DataUnavailable = 8
Expected data is unavailable for the requested time range due to an unmounted volume an off-line archive or similar reason for temporary unavailability.

InvalidConfig = 9
The configuration for the session contains an error

Unspecified = 255
Unspecified error. See log file for more information.

class PyPlcnextRsc.Arp.Services.DataLogger.Services.**RecordType**(*value*)
Attribute to mark the recorded values of a triggered session

NONE = 0
Initialization value

Continuous = 1
Record belongs to continously logging session.

PreCycle = 2
Values are recorded before the condition was triggered

Trigger = 3
Records are recorded when the condition was triggered

PostCycle = 4
Records are recorded after the condition was triggered

class PyPlcnextRsc.Arp.Services.DataLogger.Services.**RpnItemType**(*value*)
Item type of the *TriggerRpnItem* structure. (RPN = Reverse Polnish Notation)

NONE = 0
Initialization value

Variable = 1
The Value of Item is the instance path of a variable.

Constant = 2
The Value of the Item is a constant.

Operation = 3
The value of Item is a byte containing a value of the *TriggerConditionOperation* enumeration.

class PyPlcnextRsc.Arp.Services.DataLogger.Services.**SessionPropertyName**(*value*)
All available names of properties that can be set on a session

Undefined = 0
Determines a newly created not yet configured property *SessionProperty* with undefined name will be ignored.

SamplingInterval = 1
The desired sampling rate. Can either be provided as Int64 which will be interpreted as microseconds count or as string containing the actual unit, e.g. "100ms". *Subscribe()* for more information about the sampling rate

PublishInterval = 2
The rate (as Uint16) in which values will be written to the session´s sink. Can either be provided as Int64 which will be interpreted as microseconds count or as string containing the actual unit, e.g. 100ms

BufferCapacity = 3
Amount of capacity of the underlying ring buffer. See *CreateRecordingSubscription()* for more information

SinkType = 4
The type of sink used by the session. Must be of type *SinkType*

SinkProperties = 5
Special property to configure a session sinks. Properties must be provided as string.

class PyPlcnextRsc.Arp.Services.DataLogger.Services.**SessionState**(*value*)
State of a data logger session

NONE = 0
Initialization value

Created = 1
The session was already created but not yet initialized

Initialized = 2
The session has loaded a configuration and is ready to run

Running = 3
The session is currently running and logging variables

Stopped = 4
The session is currently not running.

Error = 5
The session is in an error state.

class PyPlcnextRsc.Arp.Services.DataLogger.Services.**SinkType**(*value*)
Enumeration of possible sink types.

NONE = 0
Value not initialized.

Empty = 1
No sink assigned to session yet.

Database = 2
Sink using SQLite based database.

Volatile = 3
Sink used to store data in volatile memory, i.e. after a power reset or a deletion of the session all data is lost.

TSDB = 4
Sink used to store data in timeseries data base.

class PyPlcnextRsc.Arp.Services.DataLogger.Services.**TriggerConditionOperation**(*value*)
The TraceController provides an Interface to manage and control the LTTng Tracing on the Control

NONE = 0
No trigger condition, start recording immediately.

Equals = 1
Start recording if TriggerVariable1 is equal to TriggerVariable2.

NotEqual = 2
Start recording if TriggerVariable1 is greater than to TriggerVariable2.

GreaterThan = 3
Start recording if TriggerVariable1 is greater than to TriggerVariable2.

GreaterEqual = 4
Start recording if TriggerVariable1 is greater or equal to TriggerVariable2.

LessThan = 5
Start recording if TriggerVariable1 is less than TriggerVariable2.

LessEqual = 6
Start recording if TriggerVariable1 is less or equal to TriggerVariable2.

Modified = 7
Start recording when a modification of the TriggerVariable1 is detected.

RisingEdge = 8
Start recording when a positive (rising) edge of the TriggerVariable1 is detected.

FallingEdge = 9
Start recording when a negative (falling) edge of the TriggerVariable1 is detected.

And = 10
Start recording if TriggerCondition1 and TriggerCondition2 is true.

Or = 11
Start recording if TriggerCondition1 or TriggerCondition2 is true.

class PyPlcnextRsc.Arp.Services.DataLogger.Services.**SessionProperty**(*Name:* PyPlcnextRsc.Arp.Services.DataLogger.Services.Sessio
Value: PyPlcnextRsc.common.objects.rsc_variant.RscVariant)

All available names of properties that can be set on a session

Name: *PyPlcnextRsc.Arp.Services.DataLogger.Services.SessionPropertyName*
Name of attribute

Value: *PyPlcnextRsc.common.objects.rsc_variant.RscVariant*
Current value of attribute

class PyPlcnextRsc.Arp.Services.DataLogger.Services.**TriggerRpnItem**(*Type:* PyPlcnextRsc.Arp.Services.DataLogger.Services.RpnIte
Item: PyPlcnextRsc.common.objects.rsc_variant.RscVariant)

Item of the trigger condition

Type: *PyPlcnextRsc.Arp.Services.DataLogger.Services.RpnItemType*
Type of item (Variable, Constant or Operation)

Item: *PyPlcnextRsc.common.objects.rsc_variant.RscVariant*
Data.

class PyPlcnextRsc.Arp.Services.DataLogger.Services.**IDataLoggerService2**
The DataLogger provides an interface to log and store variables during firmware runtime.

for more information:

https://www.plcnext.help/te/Service_Components/Remote_Service_Calls_RSC/RSC_IDataLoggerService2.htm

ListSessionNames() → Tuple[str]
List all names of sessions inside the data logger component. :return: Array of session names. :rtype: Tuple[str]

CreateSession(*sessionName: str*, *persistent: bool = False*) → *PyPlcnextRsc.Arp.Services.DataLogger.Services.ErrorCode*
Tries to create a new session.

Parameters

- **sessionName** (*str(max=512)*) – Name of session to be created.
- **persistent** (*bool*) – If set to *true*, the newly created session will not be removed when the RSC connection is closed.

Returns *ErrorCode* for more information

Return type *ErrorCode*

RemoveSession(*sessionName: str*) → *PyPlcnextRsc.Arp.Services.DataLogger.Services.ErrorCode*
Tries to remove a session.

Parameters **sessionName** (*str(max=512)*) – Name of session to be removed.

Returns *ErrorCode* for more information

Return type *ErrorCode*

StartSession(*sessionName: str*) → *PyPlcnextRsc.Arp.Services.DataLogger.Services.ErrorCode*
Tries to start a logging session.

Parameters **sessionName** (*str(max=512)*) – Name of session to be started.

Returns *ErrorCode* for more information

Return type *ErrorCode*

StopSession(*sessionName: str*) → *PyPlcnextRsc.Arp.Services.DataLogger.Services.ErrorCode*
Tries to stop a logging session.

Parameters **sessionName** (*str(max=512)*) – Name of session to be stopped.

Returns *ErrorCode* for more information

Return type *ErrorCode*

ConfigureSession(*sessionName: str*, *properties: Sequence[*PyPlcnextRsc.Arp.Services.DataLogger.Services.SessionProperty*]*) → *PyPlcnextRsc.Arp.Services.DataLogger.Services.ErrorCode*
(Re)configures a session

Parameters

- **sessionName** (*str(max=512)*) – Name of session to be created or reconfigured
- **properties** (*Sequence[SessionProperty]*) – Collection of attributes forming the configuration for the session.

Returns *ErrorCode* for more information

Return type *ErrorCode*

GetSessionConfiguration(*sessionName: str*) → Tuple[Tuple[*PyPlcnextRsc.Arp.Services.DataLogger.Services.SessionProperty*], bool, *PyPlcnextRsc.Arp.Services.DataLogger.Services.ErrorCode*]
Tries to query the current configuration of a session

Parameters **sessionName** (*str(max=512)*) – Name of session to be queried

Returns

tuple with 3 return values :

- properties: Collection of attributes forming the sessions configuration after successfull invocation
- isPersistent: Determines if the session will remain (<c>true</c>) when the connection to the server is closed or not
- ErrorCode: *ErrorCode* for more information

Return type tuple[tuple[*SessionProperty*],bool,*ErrorCode*]

GetSessionState(*sessionName: str*) → Tuple[*PyPlcnextRsc.Arp.Services.DataLogger.Services.SessionState*, *PyPlcnextRsc.Arp.Services.DataLogger.Services.ErrorCode*]

Tries to query the state of a session.

Parameters **sessionName** (*str(max=512)*) – Name of session to query state of.

Returns

tuple with 2 return values :

- state: Container for state of session, if session exists. The value after return from call is unspecified if the session does not exists.
- ErrorCode: *ErrorCode* for more information

Return type tuple[*SessionState*,*ErrorCode*]

SetVariables(*sessionName: str*, *variableNames: Sequence[str]*) → Tuple[*PyPlcnextRsc.Arp.Services.DataLogger.Services.ErrorCode*]

Tries to add a variable to a session.

Parameters

- **sessionName** (*str(max=512)*) – Name of session where variable should be added.
- **variableNames** (*Sequence[str(max=512)]*) – Name of session where variable should be added.

Returns Returns a tuple of *ErrorCode*, in the same order as the variables were added.

Return type tuple[*ErrorCode*]

GetLoggedVariables(*sessionName: str*) → Tuple[Tuple[*PyPlcnextRsc.Arp.Plc.Gds.Services.VariableInfo*], *PyPlcnextRsc.Arp.Services.DataLogger.Services.ErrorCode*]

Queries all infos about logged variables of a session.

Parameters **sessionName** (*str(max=512)*) – Name of session to query logged variables

Returns

tuple with 2 return values :

- infos: tuple which list logged variables after successful call.
- ErrorCode: *ErrorCode* for more information

Return type tuple[tuple[*VariableInfo*],*ErrorCode*]

ReadVariablesData(*sessionName: str*, *startTime: datetime.datetime*, *endTime: datetime.datetime*, *variableNames: Sequence[str]*) → Tuple[PyPlcnextRsc.common.types.rsc_types.RscEnumerator[*PyPlcnextRsc.common.objects.rsc_variant.Rs*
PyPlcnextRsc.Arp.Services.DataLogger.Services.ErrorCode]

Read the data from the given variable from the session with the session name.

This service function returns the plain data values from the passed variable names including timestamps and data series consistent flags, which is called a record. In a record the values are in a static order and doesn't contain any type information. Each record starts with the timestamp followed by the values from the given variable by names and the consistent flag. The record ends with a record type describing the cycle the record belongs to.

Example:

Read variables from task A: a1, a2 from task B: b1

Results in: object[] timestamp task A, a1, a2, b1, consistent flag, record type timestamp task B, a1, a2, b1, consistent flag, record type

The number of records depends on the given start and end time. Each values will be returned between the start and end time. If the start time is zero, all available records until the end time will be returned.

If the end time is zero, all available records from the start time until the last available record is reached will be returned.

If the start and end time is zero, each available record will be returned.

If the start time is greater than the end time, the resulted values are returned in descending order.

Parameters

- **sessionName** (*str(max=512)*) – Name of session where variable should be read from.
- **startTime** (*datetime*) – Start time to be read data.
- **endTime** (*datetime*) – End time to be read data.
- **variableNames** (*Sequence[str(max=512)]*) – Name of variables to be read data.

Returns

tuple with 2 return values :

- values: tuple which stores the read values.
- ErrorCode: *ErrorCode* for more information

Return type tuple[tuple[*RscVariant*],*ErrorCode*]

GetRotatedFileNames(*sessionName: str*) → Tuple[Tuple[str], *PyPlcnextRsc.Arp.Services.DataLogger.Services.ErrorCode*]

Returns names of all files that have been written by a session

Parameters **sessionName** (*str(max=512)*) – Name of session from which rotated files should be listed

Returns

tuple with 2 return values :

- filenames: list names of all rotated files on successful call.
- ErrorCode: *ErrorCode* for more information

Return type tuple[tuple[*VariableInfo*],*ErrorCode*]

GetSessionNames(*variableName: str*) → Tuple[str]
Tries to retrieve names of sessions which log assigned variables

Parameters **variableName** (*str(max=512)*) – Name of variable to which corresponding sessions should be found

Returns Tuple of names of sessions which log the variable in question

Return type tuple[str(max=512)]

SetTriggerCondition(*sessionName: str*, *taskName: str*, *preCycleCount: int*, *postCycleCount: int*, *triggerCondition: Sequence[*PyPlcnextRsc.Arp.Services.DataLogger.Services.TriggerRpnItem*]*) → *PyPlcnextRsc.Arp.Services.DataLogger.Services.ErrorCode*
Sets a trigger condition

Configuration of the trigger is done in RPN (Reverse Polish Notation). Each operand or operation is a single *TriggerRpnItem*. Only if the condition specified by the trigger is fullfilled values are logged and stored inside the sink.

The amount of cycles that should be stored before the condition was fullfilled can be configured using *preCycleCount* whereas the amount of cycles that should be recorded afterwards can be configured using *postCycleCount*. If *postCycleCount* is set to 0 then the recording continues until IDataLoggerService::StopSession is called or the PLC project is stopped.

Parameters

- **sessionName** (*str(max=512)*) – Name of session to set trigger condition
- **taskName** (*str(max=512)*) – Name of task where trigger condition is evaluated
- **preCycleCount** (*uint16*) – Amount of datasets recorded before the condition was triggered
- **postCycleCount** (*uint16*) – Amount of dataset recorded after the condition is triggered (0 means endless)
- **triggerCondition** (*Sequence[TriggerRpnItem]*) – List of trigger items. All items are evaluated in order of their position inside the list.

Returns Returns a tuple of *ErrorCode*, in the same order as the variables were added.

Return type tuple[*ErrorCode*]

2.4.3 Arp.System.Commons.Services.Io

class PyPlcnextRsc.Arp.System.Commons.Services.Io.**FileSystemError**(*value*)
This enum is used by several file operations.

NONE = 0
Success

Unknown = 1
The error is not listed in this enumeration.

InvalidPath = 2
The path is invalid.

NotExist = 3
The path does not exist.

AlreadyExists = 4
The path already exists.

AccessDenied = 5
The file is already in use.

OutOfSpace = 6
There is not enough space on the Device left.

class PyPlcnextRsc.Arp.System.Commons.Services.Io.**Permissions**(*value*)
Provides attributes for files and directories.

NONE = 0
NoPerms of posix

OthersExe = 1
Execute/search permission, owner

OthersWrite = 2
Write permission, owner

OthersRead = 4
Read permission, owner

OthersAll = 7
Read, Write, execute/search by owner; owner_read | owner_write | owner_exe

GroupExe = 8
Execute/search permission, group

GroupWrite = 16
Execute/search permission, group

GroupRead = 32
Read permission, group

GroupAll = 56
Read, Write, execute/search by group; group_read | group_write | group_exe

OwnerExe = 64
Read permission, others

OwnerWrite = 128
Write permission, others

OwnerRead = 256
Execute/search permission, others

OwnerAll = 448
Read, Write, execute/search by others; others_read | others_write | others_exe

AllAll = 511
owner_all | group_all | others_all

class PyPlcnextRsc.Arp.System.Commons.Services.Io.**Traits**(*value*)
This enum is used by several file Services to specify the file traits to get or set, respectively.

NONE = 0
Not set.

Permissions = 1
Specifies the file permissions mask as Arp.System.Commons.Services.Io.Permissions mask.

LastWriteTime = 2
Specifies the time of last Write access or last modified time, respectively as System.DateTime in UTC

Length = 4
Specifies the size of the file in bytes as System.Int64.

Crc32 = 8
Specifies the CRC32 value of the file as System.Int32.

class PyPlcnextRsc.Arp.System.Commons.Services.Io.**TraitItem**(*Trait:* PyPlcnextRsc.Arp.System.Commons.Services.Io.Traits, *Value:* PyPlcnextRsc.common.objects.rsc_variant.RscVariant)

Specifies a file trait item

Trait: *PyPlcnextRsc.Arp.System.Commons.Services.Io.Traits*
The file trait of this item.

Value: *PyPlcnextRsc.common.objects.rsc_variant.RscVariant*
The value of the file info trait of this item.

class PyPlcnextRsc.Arp.System.Commons.Services.Io.**FileSystemEntry**(*Path: str*, *IsFile: bool*, *IsDirectory: bool*)

This struct is used by file operations to reading of the file system entries.

Path: str
The path of the file system entry (file or directory)

IsFile: bool
The specified path ia a file.

IsDirectory: bool
The specified path is a directory.

class PyPlcnextRsc.Arp.System.Commons.Services.Io.**FileSystemTraitsEntry**(*Path: str*, *IsFile: bool*, *IsDirectory: bool*, *Traits: Sequence[*PyPlcnextRsc.Arp.System.Comm

This struct is used by file operations reading file informations from Device.

Path: str
The path of the file.

IsFile: bool
The specified path ia a file.

IsDirectory: bool
The specified path ia a directory.

Traits: Sequence[*PyPlcnextRsc.Arp.System.Commons.Services.Io.TraitItem*]
The requested file traits of the file.

class PyPlcnextRsc.Arp.System.Commons.Services.Io.**SpaceInfo**(*Capacity: int*, *Free: int*, *Available: int*)

This struct is used by file operations to reading of the space information.

Capacity: int
Capacity space of the Device.

Free: int
Free space of the Device.

Available: int
Available space of the Device.

class PyPlcnextRsc.Arp.System.Commons.Services.Io.**IFileSystemInfoService**
A generic service to retrieve file system infos.

GetSupportedTraits() → *PyPlcnextRsc.Arp.System.Commons.Services.Io.Traits*
Gets the supported traits.

Returns The supported traits as bitset.

Return type *Traits*

GetPermissions(*path: str*) → Tuple[*PyPlcnextRsc.Arp.System.Commons.Services.Io.Permissions*, *PyPlcnextRsc.Arp.System.Commons.Services.Io.FileSystemError*]
Gets the permissions of the specified path.

Parameters **path** (*str(max=512)*) – The path to get the permissions from.

Returns

tuple with 2 return values :

0. The permissions of the specified path.
1. Result of the action.

Return type tuple[*Permissions*, *FileSystemError*]

AddPermissions(*path: str*, *permissions:* PyPlcnextRsc.Arp.System.Commons.Services.Io.Permissions) → *PyPlcnextRsc.Arp.System.Commons.Services.Io.FileSystemError*
Adds the permissions to the specified path.

Parameters

- **path** (*str(max=512)*) – The path to get the permissions from.
- **permissions** (Permissions) – The permissions to add.

Returns Result of the action.

Return type *FileSystemError*

RemovePermissions(*path: str*, *permissions:* PyPlcnextRsc.Arp.System.Commons.Services.Io.Permissions) → *PyPlcnextRsc.Arp.System.Commons.Services.Io.FileSystemError*
Removes the permissions of the specified path.

Parameters

- **path** (*str(max=512)*) – The path to get the permissions from.
- **permissions** (Permissions) – The permissions to remove.

Returns Result of the action.

GetFileSystemTraitsEntry(*traits:* PyPlcnextRsc.Arp.System.Commons.Services.Io.Traits, *path: str*) → Tuple[*PyPlcnextRsc.Arp.System.Commons.Services.Io.FileSystemTraitsEntry*, *PyPlcnextRsc.Arp.System.Commons.Services.Io.FileSystemError*]
Gets the file system traits entry of the specified path.

Parameters

- **traits** (Traits) – The selection of traits to get.
- **path** (*str(max=512)*) – The path to get the file system traits entry from.

Returns

tuple with 2 return values :

0. The file system traits entry of the specified path
1. Result of the action.

Return type tuple[*FileSystemTraitsEntry*,*FileSystemError*]

GetSpaceInfo(*path: str*) → Tuple[*PyPlcnextRsc.Arp.System.Commons.Services.Io.SpaceInfo*, *PyPlcnextRsc.Arp.System.Commons.Services.Io.FileSystemError*]

Gets the space information of the specified path.

Parameters **path** (*str(max=512)*) – The path to get the file system traits entry from.

Returns

tuple with 2 return values :

0. The space information of the specified path.
1. Result of the action.

Return type tuple[*SpaceInfo*, *FileSystemError*]

GetRootDirectories() → Tuple[str]

Gets a list of all root directories supported by the target.

Returns A list of all root directories.

Return type tuple[str]

class PyPlcnextRsc.Arp.System.Commons.Services.Io.**IFileService**

Provides common file operations for reading and writing files as well as deleting/moving/copying files on the Device.

Exists(*path: str*) → bool

Checks if the specified file exists.

Returns true if the file exists, otherwise false

Return type bool

Write(*filePath: str*, *overwrite: bool*, *traitItems: Sequence[*PyPlcnextRsc.Arp.System.Commons.Services.Io.TraitItem*]*, *data:* PyPlcnextRsc.common.objects.rsc_stream.RscStream) → *PyPlcnextRsc.Arp.System.Commons.Services.Io.FileSystemError*

Writes the given data to the specified file.

Parameters

- **filePath** (*str(max=512)*) – Path of the file on the target.
- **overwrite** (*bool*) – If set to true the destination file is overwritten, if it yet exists, otherwise an error is returned.
- **traitItems** (*Sequence[*TraitItem*]*) – Trait items to set up after writing the file.
- **data** (RscStream) – Data to writing into the specified file.

Returns Result of the action.

Return type *FileSystemError*

Read(*fileTraits:* PyPlcnextRsc.Arp.System.Commons.Services.Io.Traits, *filePath: str*) → Tuple[*PyPlcnextRsc.common.objects.rsc_stream.RscStream*, Tuple[*PyPlcnextRsc.Arp.System.Commons.Services.Io.TraitItem*], *PyPlcnextRsc.Arp.System.Commons.Services.Io.FileSystemError*]

Reads the specified file from Device.

Parameters

- **fileTraits** (*Traits*) – Specifies the file traits to Read, if this value is not *PyPlcnextRsc.Arp.System.Commons.Services.Io.Traits.NONE*
- **filePath** (*str(max=512)*) – The path of the file to Read

Returns

tuple with 3 return values :

0. Data Read from the specified file.
1. Specified trait items Read from the specified file.
2. Result of the action.

Return type tuple[*RscStream*,tuple[*TraitItem*],*FileSystemError*]

Delete(*filePath: str*) → *PyPlcnextRsc.Arp.System.Commons.Services.Io.FileSystemError*

Deletes the specified file.

Parameters **filePath** (*str(max=512)*) – The path of the file to delete.

Returns Result of the action.

Return type *FileSystemError*

Move(*createDirectory: bool*, *overwrite: bool*, *sourceFilePath: str*, *destinationFilePath: str*) → *PyPlcnextRsc.Arp.System.Commons.Services.Io.FileSystemError*

Moves the specified file.

Parameters

- **createDirectory** (*bool*) – if set to true the directory of the file is created (recursively), if it does not exists yet.
- **overwrite** (*bool*) – if set to true the destination file is overwritten, if it yet exists, otherwise an error is returned.
- **sourceFilePath** (*str(max=512)*) – The source path of the file to move.
- **destinationFilePath** (*str(max=512)*) – The destination path of the file to move.

Returns Result of the action.

Return type *FileSystemError*

Copy(*createDirectory: bool*, *overwrite: bool*, *sourceFilePath: str*, *destinationFilePath: str*) → *PyPlcnextRsc.Arp.System.Commons.Services.Io.FileSystemError*

Copies the specified files.

Parameters

- **createDirectory** (*bool*) – if set to true the directory of the file is created (recursively), if it does not exists yet.
- **overwrite** (*bool*) – If set to true the destination file is overwritten, if it yet exists, otherwise an error is returned.
- **sourceFilePath** (*str(max=512)*) – The source path of the file to copy.

- **destinationFilePath** (*str(max=512)*) – The destination path of the file to copy.

Returns Result of the action.

Return type *FileSystemError*

class PyPlcnextRsc.Arp.System.Commons.Services.Io.**IDirectoryService**

Provides common file directory operations.

Exists(*path: str*) → bool

Checks if the specified directory exists.

Parameters **path** (*str(max=512)*) – The path of the directory to check.

Returns true if the directory exists, otherwise false.

Return type bool

Create(*path: str*) → *PyPlcnextRsc.Arp.System.Commons.Services.Io.FileSystemError*

Creates the specified directory.

Parameters **path** (*str(max=512)*) – The path of the directory to create.

Returns Result of the action.

Return type *FileSystemError*

Delete(*path: str*) → *PyPlcnextRsc.Arp.System.Commons.Services.Io.FileSystemError*

Deletes the specified directory and its content.

Parameters **path** (*str(max=512)*) – The path of the directory to delete.

Returns Result of the action.

Return type *FileSystemError*

Clear(*path: str*) → *PyPlcnextRsc.Arp.System.Commons.Services.Io.FileSystemError*

Removes the content of the specified directory, but does not delete the specified directory itself.

Parameters **path** (*str(max=512)*) – The path of the directory to clear.

Returns Result of the action.

Return type *FileSystemError*

Move(*sourcePath: str*, *destinationPath: str*, *clear: bool = False*) → *PyPlcnextRsc.Arp.System.Commons.Services.Io.FileSystemError*

Moves the specified directory and its content to the given new location.

Parameters

- **sourcePath** (*str(max=512)*) – The source path of the directory to move.
- **destinationPath** (*str(max=512)*) – The destination path of the directory to move all content to.
- **clear** (*bool*) – If set to true the destination location is cleared first if it yet exists and the operation succeeds anyway while returning true. Otherwise, if the destination yet exists, the operations fails and returns false.

Returns Result of the action.

Return type *FileSystemError*

Copy(*sourcePath: str*, *destinationPath: str*, *clear: bool = False*) → *PyPlcnextRsc.Arp.System.Commons.Services.Io.FileSystemError*

Copies the specified directory and its content to the given new location.

Parameters

- **sourcePath** (*str(max=512)*) – The source path of the directory to copy.
- **destinationPath** (*str(max=512)*) – The destination path of the directory to copy all content to.
- **clear** (*bool*) – If set to true the destination location is cleared first if it yet exists and the operation succeeds anyway while returning true. Otherwise, if the destination yet exists, the operations fails and returns false.

Returns Result of the action.

Return type *FileSystemError*

EnumerateFileSystemEntries(*path: str*, *searchPattern: str*, *recursive: bool*) → PyPlcnextRsc.common.types.rsc_types.RscEnumerator[*PyPlcnextRsc.Arp.System.Commons.Services.Io*

Enumerates all files and subdirectories of the specified directory.

Parameters

- **path** (*str(max=512)*) – The path of the directory to search in.
- **searchPattern** (*str(max=512)*) – The pattern of the files to enumerate.
- **recursive** (*bool*) – If set to true the files of all subdirectories are liested as well.

Returns A file system entry for each found file.

Return type list[*FileSystemEntry*]

EnumerateFileSystemTraitsEntries(*path: str*, *searchPattern: str*, *traits:* PyPlcnextRsc.Arp.System.Commons.Services.Io.Traits, *recursive: bool*) → PyPlcnextRsc.common.types.rsc_types.RscEnumerator[*PyPlcnextRsc.Arp.System.Commons.Se*

Enumerates all files and subdirectories of the specified directory.

Parameters

- **path** (*str(max=512)*) – The path of the directory to search in.
- **searchPattern** (*str(max=512)*) – The pattern of the files to enumerate.
- **traits** (Traits) – The selection of traits to get.
- **recursive** (*bool*) – if set to true the files of all subdirectories are listed as well.

Returns A file system trait entry for each found file.

Return type list[*FileSystemTraitsEntry*]

2.4.4 Arp.Plc.Gds.Services

Namespace for services of the Global Data Space (GDS)

class PyPlcnextRsc.Arp.Plc.Gds.Services.**DataType**(*value*)

NONE = 0
Unspecified.

Void = 1
Void - Arp C++ empty type

Bit = 2
Bit - Arp C++ data type (1 Byte)

Boolean = 3
Boolean - Arp C++ data type (1 Byte)

UInt8 = 4
UInt8 - Arp C++ data type (1 Byte)

Int8 = 5
Int8 - Arp C++ data type (1 Byte)

Char8 = 6
Char8 - Arp C++ data type (1 Byte)

Char16 = 7
Char16 - Arp C++ data type (2 Byte)

UInt16 = 8
UInt16 - Arp C++ data type (2 Byte)

Int16 = 9
Int16 - Arp C++ data type (2 Byte)

UInt32 = 10
UInt32 - Arp C++ data type (4 Byte)

Int32 = 11
Int32 - Arp C++ data type (4 Byte)

UInt64 = 12
UInt64 - Arp C++ data type (8 Byte)

Int64 = 13
Int64 - Arp C++ data type (8 Byte)

Float32 = 14
Float32 - Arp C++ data type (4 Byte)

Float64 = 15
Float64 - Arp C++ data type (8 Byte)

Primitive = 32
Limit of primitive types

DateTime = 33
C++ DateTime type

IecTime = 34
IEC type: TIME [int32]

IecTime64 = 35
IEC type: LTIME [int64]

IecDate = 36
IEC type: DATE [N/A],Not supported by PCWE.

IecDate64 = 37
IEC type: LDATE [int64]

IecDateTime = 38
IEC type: DATE_AND_TIME, DT [N/A],Not supported by PCWE.

IecDateTime64 = 39
IEC type: LDATE_AND_TIME, LDT [int64]

IecTimeOfDay = 40
IEC type: TIME_OF_DAY, TOD [N/A],Not supported by PCWE.

IecTimeOfDay64 = 41
IEC type: LTIME_OF_DAY, LTOD [int64]

StaticString = 42
Static String type

IecString = 43
Iec String type, only for internal use

ClrString = 44
.NET/C# String type, only for internal use

String = 45
C++ String type, only for internal use

Elementary = 64
Limit of elementary types.

ArrayElement = 65
ArrayOfArray

Struct = 66
Struct

Class = 67
Class

FunctionBlock = 68
Function Block

Subsystem = 69
Subsystem

Program = 70
Program

Component = 71
Component

Library = 72
Library

Complex = 254
Limit of complex types

Pointer = 512
Determines a pointer type.Pointer are declared as *PyPlcnextRsc.Arp.Plc.Commons.DataType.Elementary* kind.

Array = 1024
Determines an array type.Arrays are declared as *PyPlcnextRsc.Arp.Plc.Commons.DataType.Elementary* kind.

Enum = 2048
Determines an Enumeration type.Enums are declared as *PyPlcnextRsc.Arp.Plc.Commons.DataType.Elementary* kind.

Reference = 4096
Determines a C# reference type.Reference are declared as *PyPlcnextRsc.Arp.Plc.Commons.DataType.Elementary* kind.

BaseTypeMask = 255
For removing all flags

class PyPlcnextRsc.Arp.Plc.Gds.Services.**DataAccessError**(*value*)
This enumeration contains the possible data access errors.

NONE = 0
No error.

NotExists = 1
The variable does not exist.

NotAuthorized = 2
The user is not authorized.

TypeMismatch = 3
During a Write operation the type of the value is not suitable for the particular variable. The *IDataAccessService* does not convert types. The type of each value which is to be written needs to be suitable for the particular variable.

PortNameSyntaxError = 4
The name of the variable as given during a Write or Read operation is syntactically not correct. For example the variable name contains an index range.

PortNameSemanticError = 5
The semantic of the name of the variable as given during a Write or Read operation is semantically not correct. For example the variable name contains an index range with a start index not lower than the end index.

IndexOutOfRange = 6
The variable name contains an index which is out of range.

NotImplemented = 7
The variable type is not implemented yet.

NotSupported = 8
The variable type is not supported.

CurrentlyUnavailable = 9
The requested service is currently not available.

UnvalidSubscription = 10
Invalid subscription.

NoData = 11
NoData available.

InvalidConfig = 12
The configuration for the subscription contains an error.

Unspecified = 255
Unspecified error. See log file for more information.

class PyPlcnextRsc.Arp.Plc.Gds.Services.**ReadItem**(*Error:* PyPlcnextRsc.Arp.Plc.Gds.Services.DataAccessError, *Value:* PyPlcnextRsc.common.objects.rsc_variant.RscVariant)
Stores the data to be Read, written by the controller and a possible data access error.

Error: *PyPlcnextRsc.Arp.Plc.Gds.Services.DataAccessError*
Contains the possible data access errors.

Value: *PyPlcnextRsc.common.objects.rsc_variant.RscVariant*
Contains the data to be Read, written by the controller.

class PyPlcnextRsc.Arp.Plc.Gds.Services.**WriteItem**(*PortName: str*, *Value:* PyPlcnextRsc.common.objects.rsc_variant.RscVariant)
Stores the to be written data and the related variable name to be Write to.

PortName: **str**
Name of the variable.

Value: *PyPlcnextRsc.common.objects.rsc_variant.RscVariant*
Contains the to be written data.

class PyPlcnextRsc.Arp.Plc.Gds.Services.**SubscriptionKind**(*value*)
This enumeration contains the possible kinds of subscriptions.

NONE = 0

HighPerformance = 1
The subscription operates with a task-triggered DoubleBuffer, which holds the last written port data.

RealTime = 2
The subscription operates with a task-triggered QuadBuffer, which holds the last written port data.

Recording = 3
The subscription operates with a task-triggered RingBuffer, which holds the last N numbers of written data.

ClosedRealTime = 4
The subscription operates with a task-triggered RingBuffer, which holds the last N numbers of written data.

DirectRead = 5
The subscription operates with a self-triggered DoubleBuffer, which holds the last written port data.

class PyPlcnextRsc.Arp.Plc.Gds.Services.**VariableInfo**(*Name: str*, *Type:* PyPlcnextRsc.Arp.Plc.Commons.DataType)
Describes a subscribed variable.

Name: **str**
Full name of the variable.

Type: *PyPlcnextRsc.Arp.Plc.Commons.DataType*
DataType 0f the variable

class PyPlcnextRsc.Arp.Plc.Gds.Services.**ForceItem**(*VariableName: str*, *ForceValue:* PyPlcnextRsc.common.objects.rsc_variant.RscVariant)
A force item structure.

VariableName: **str**
The instance path of the forced item.

ForceValue: *PyPlcnextRsc.common.objects.rsc_variant.RscVariant*
The value of the forced item.

class PyPlcnextRsc.Arp.Plc.Gds.Services.**IDataAccessService**
Services for the direct data access. The direct access functionality is a way for reading and writing values from and to variables. This is the fastest way, with a minimum of influence to the real time process, but it is not guaranteed that the data will be Read/Write in the same task cycle. For task consistent reading the subscription service *ISubscriptionService* has to be used. A client can Read/Write from/to different types of variables provided in *PyPlcnextRsc.Arp.Plc.Commons.DataType*. Currently supported types are listed below:

Type	Supported	Description
PyPlcnextRsc.Arp.Plc.Commons.DataType.Primitive	YES	.
PyPlcnextRsc.Arp.Plc.Commons.DataType.DateTime	YES	.
PyPlcnextRsc.Arp.Plc.Commons.DataType.String	(YES)	Please use *StaticString* or *IecString*
PyPlcnextRsc.Arp.Plc.Commons.DataType.Enum	YES	
PyPlcnextRsc.Arp.Plc.Commons.DataType.Struct	YES	
PyPlcnextRsc.Arp.Plc.Commons.DataType.FunctionBlock	YES	
PyPlcnextRsc.Arp.Plc.Commons.DataType.Pointer	YES	
PyPlcnextRsc.Arp.Plc.Commons.DataType.Array	YES	

To address a variable, the full variable name uri is necessary. Some valid examples are given below:

- ComponentName-1/ProgramName-1.Variable_Name
- ComponentName-1/Global_Variable_Name
- ComponentName-1/ProgramName-1.Array_Variable_Name
- ComponentName-1/ProgramName-1.Array_Variable_Name[index]
- ComponentName-1/ProgramName-1.Array_Variable_Name[startIndex:endIndex]
- ComponentName-1/ProgramName-1.Struct_Variable_Name.Element1.Leaf
- ComponentName-1/ProgramName-1.Struct_Variable_Name.Element1.LeafArray
- ComponentName-1/ProgramName-1.Struct_Variable_Name.Element1.LeafArray[index]

ReadSingle(*portName: str*) → *PyPlcnextRsc.Arp.Plc.Gds.Services.ReadItem*

Reads the value of the variable directly from the given variable name.

Copies the value of the variable, given by the variable name, to the *ReadItem* object, which will be returned. ReadSingle can only Read one single variable, so if you want to Read multiple variables simultaneously, an array or a range of an array, you have to use the *Read()* service instead. Be aware, this copy process isn't task consistent and the data could be corrupted.

Parameters **portName** (*str(max=512)*) – Full variable name uri.

Returns see *ReadItem*

Return type *ReadItem*

Read(*portNames: Sequence[str]*) → Tuple[*PyPlcnextRsc.Arp.Plc.Gds.Services.ReadItem*]

Reads the variable values directly from the given variable names. Copies the values of the variables, given by the variable names, to a vector of *ReadItem* objects, which will be returned. Be aware, this copy process isn't task consistent and the data could be corrupted.

Parameters **portNames** (*Sequence[str(max=512)]*) – An array of full variable name uris.

Returns Returns a tuple of *ReadItem*, in the same order as the variables were added.

Return type tuple[*ReadItem*]

WriteSingle(*data:* PyPlcnextRsc.Arp.Plc.Gds.Services.WriteItem) → *PyPlcnextRsc.Arp.Plc.Gds.Services.DataAccessError*

Writes the given value to the given variable name containing in the given Arp.Plc.Gds.Services.WriteItem.

Writes the given value to the given variable containing in the given *WriteItem* object. WriteSingle can only Write one single value, so if you want to Write to multiple variables simultaneously, to an array or to a range of an array, you have to use the *Write()* service instead. Be aware, this Write process isn't task consistent and the data could be corrupted.

Parameters **data** (WriteItem) – Variable data which contains the variable name and the value to be written.

Returns Returns *PyPlcnextRsc.Arp.Plc.Gds.Services.DataAccessError.NONE* on success.

Return type *DataAccessError*

Write(*writeData: Sequence[*PyPlcnextRsc.Arp.Plc.Gds.Services.WriteItem*]*) → Tuple[*PyPlcnextRsc.Arp.Plc.Gds.Services.DataAccessError*]

Writes the given values to the given variables containing in the given *WriteItem* objects.

Writes the given values to the given variables containing in the given *WriteItem* objects. Be aware, this Write process isn't task consistent and the data could be corrupted.

Parameters **writeData** (*Sequence[*WriteItem*]*) – Array of *WriteItem*, which contains the variable name and the value to be written.

Returns Returns a vector of *DataAccessError*, *PyPlcnextRsc.Arp.Plc.Gds.Services.DataAccessError.NONE* on success, in the same order as the variables were added.

Return type tuple[*DataAccessError*]

class PyPlcnextRsc.Arp.Plc.Gds.Services.**ISubscriptionService**

Services for the subscription.

The subscription functionality is a more elegant way reading values from variables, in contrast to permanently reading (polling). A client can subscribe a selection of variables of interest and the subscription will copy the data values to a internalEnums buffer. This is the recommended mechanism to "Read" variable values from th PLC system. All Read data are always task consistent, because the data is written by the task itself. The data updating rate depends of the task to which the variable belongs. Because global variables haven't a task affiliation, each global variable will be updated by default from the task 'Globals'. This task has a default cycling time of 50ms which is configurable via the ESM configuration. For more information to the update rate see Arp.Plc.Gds.Services.ISubscriptionService.Subscribe. Initially, the internalEnums buffers are initialized with null values (*PyPlcnextRsc.Arp.Plc.Commons.DataType.Void*). This is important to know, especially if values are Read immediately after the subscription is created. More precisely if the data are Read before the tasks have written the data. Additionally a subscription is able to generate timestamps which will be generated at the end of the variable source task. *ReadTimeStampedValues()* A client can subscribe to different types of variables provided in *DataType*. Currently supported variable types are listed below:

Type	Supported	Description
PyPlcnextRsc.Arp.Plc.Commons.DataType.Primitive	YES	.
PyPlcnextRsc.Arp.Plc.Commons.DataType.DateTime	YES	.
PyPlcnextRsc.Arp.Plc.Commons.DataType.String	(YES)	Please use *StaticString* or *IecString*
PyPlcnextRsc.Arp.Plc.Commons.DataType.Enum	YES	
PyPlcnextRsc.Arp.Plc.Commons.DataType.Struct	YES	
PyPlcnextRsc.Arp.Plc.Commons.DataType.FunctionBlock	YES	
PyPlcnextRsc.Arp.Plc.Commons.DataType.Pointer	YES	
PyPlcnextRsc.Arp.Plc.Commons.DataType.Array	YES	

To address a variable, the full variable name (uri) is necessary. Some valid examples are given below:

- ComponentName-1/ProgramName-1.Variable_Name
- ComponentName-1/Global_Variable_Name
- ComponentName-1/ProgramName-1.Array_Variable_Name[index]
- ComponentName-1/ProgramName-1.Array_Variable_Name[startIndex:endIndex]
- ComponentName-1/ProgramName-1.Struct_Variable_Name.Element1.Leaf
- ComponentName-1/ProgramName-1.Struct_Variable_Name.Element1.LeafArray[index]
- ComponentName-1/ProgramName-1.Struct_Variable_Name.Element1.LeafArrayOfArray[indexX][indexY]

A Subscription can created in the PLC process, in other processes and also on remote targets. All subscriptions will be removed after a *Cold*, *Warm* and after a download change. While a download change is in process the subscription service will be disabled and each function will be return the error code *PyPlcnextRsc.Arp.Plc.Gds.Services.DataAccessError.CurrentlyUnavailable*.

CreateSubscription(*kind:* PyPlcnextRsc.Arp.Plc.Gds.Services.SubscriptionKind) → int

Creates a subscription of the given *SubscriptionKind*.

This method allows other components, also from remote targets, to create a subscription which is able to subscribe each PLC variable. On success it returns a unique SubscriptionId which is created internally. The SubscriptionId has to be exposed to the SDK user, due to the usage on remote targets. The SubscriptionId is the reference to a created subscription at the PLC target and is needed in each subscription method exclude this and `CreateRecordingSubscription()`. Each subscription contains at least one buffer which kind depends on the *SubscriptionKind*. The number of buffer depends on the different tasks which contains the added variables. The buffer are initialized with a *DataType* specific initial value e.g.: int8 = 0 or boolean = false. Apart from *DirectRead*, the buffer will filled by the task. How often the task stores the data to the buffer depends on the task cycle time and the configured subscription sample interval.

The *SubscriptionKind* decides which kind of a subscription will be created. Each kind has its own benefits and differs in consistence, performance and memory usage. The available kinds are listed below:

SubscriptionKind.DirectRead: The subscription itself triggers the copy process and will Read the data directly from the source. This could be the fastest way and with no influence to the real time, to get the current data, but the task consistency is not guaranteed.

Usage example: Asynchronous data collection for non critical data.

SubscriptionKind.HighPerformance: This subscription uses a DoubleBuffer which contains the last written data from the added variables. This kind is task consistent, has low influence to the real time and is low in memory usage.

Usage example: Standard way to collect the variable data.

SubscriptionKind.RealTime: This subscription uses a QuadBuffer which contains the last written data from the added variables. This kind is task consistent, has the fastest access to the current written data, but uses the fourfold memory.

Usage example: For variables which are running in high speed tasks and for which it is necessary to guarantee the fastest access to the current written data.

Note: In most cases the *HighPerformance* is sufficient.

SubscriptionKind.Recording: This subscription uses a RingBuffer which is able to store more than one record of data. This kind is task consistent, has low influence to the real time, but needs, dependent to the ring capacity, a lot of memory. By default the ring capacity is 10, use *CreateRecordingSubscription()* to create a subscription witch a self defined size.

Usage example: For variables which are running in faster tasks than the consumer does and for which it is necessary to guarantee that every data record will be stored, without a single gap.

Note: This kind uses a lot of memory!

After the subscription is created successfully, variables could be added with *AddVariable()* or *AddVariables()* with the SubscriptionId just returned in this method.

Parameters **kind** (SubscriptionKind) – The kind of the subscription.

Returns The unique subscription id on success, otherwise 0.

Return type int(Uint32)

CreateRecordingSubscription(*recordCount: int*) → int

Creates a subscription of *Recording*.

This method creates a subscription of the kind *Recording*. Compared to the method *CreateSubscription()*, it allows to configure the capacity of the internalEnums used ring buffer. For further information see *CreateSubscription()*.

Parameters **recordCount** (*int(Uint16)*) – The maximum number of storable records.

Returns The unique subscription id on success, otherwise 0.

Return type int(Uint32)

AddVariable(*subscriptionId: int*, *variableName: str*) → *PyPlcnextRsc.Arp.Plc.Gds.Services.DataAccessError*

Extends the subscription with the given id by inserting the given variable name

The added variable is stored in a internalEnums container and will be subscribed after calling *Subscribe()*. If the subscription has already been subscribed, it is necessary to call *Resubscribe()* to subscribe the new added variable finally. If the same full variable name is added multiple times, the old variable will be overridden. In case, a variable name is invalid or doesn't exists a specific error code will be returned DataAccessError, on success the code *NONE* of the added variable will be returned. A variable which doesn't returned with *PyPlcnextRsc.Arp.Plc.Gds.Services.DataAccessError.NONE* won't be added to the subscription and won't be subscribed.

A single array element can added with its index in brackets e.g.:

- ComponentName-1/ProgramName-1.Array_Name[index]

Or a rage of an array can added with tow indexes separated with a colon in brackets e.g.:

- ComponentName-1/ProgramName-1.Array_Name[StartIndex:EndIndex]

If an array variable is added without a variable specification, the entire array will be added to the subscription. An alternative way to insert variables to the subscription is by using the function *AddVariables()*.

Parameters

- **subscriptionId** (*int(Uint32)*) – The id of the subscription where the variable is add to.
- **variableName** (*str(max=512)*) – The full name of the variable.

Returns Returns *NONE* on success.

Return type *DataAccessError*

AddVariables(*subscriptionId: int*, *variableNames: Sequence[str]*) → Tuple[*PyPlcnextRsc.Arp.Plc.Gds.Services.DataAccessError*]

Extends the subscription with the given id by inserting a range of new variables.

Allows to add a range of variables to the subscription. The returned array of type *DataAccessError*, is in the same order as the given array of variable names and indicates if the given variables are valid and exist. For further information see *AddVariable()*.

Parameters

- **subscriptionId** (*int(Uint32)*) – The id of the subscription where the variable is add to.
- **variableNames** (*Sequence[str(max=512)]*) – An array of full variable names.

Returns Returns a tuple of *DataAccessError*, *NONE* on success, in the same order as the variables were added.

Return type tuple[*DataAccessError*]

RemoveVariable(*subscriptionId: int*, *variableName: str*) → *PyPlcnextRsc.Arp.Plc.Gds.Services.DataAccessError*

Removes the variable with the specific variable name from the subscription.

Removes the variable that compare equal to the given variable name, from the internalEnums variable list. If the subscription has already been subscribed, it is necessary to call *Resubscribe()* to remove the given variable from the internalEnums created buffer.

Parameters

- **subscriptionId** (*int(Uint32)*) – The id of the subscription.
- **variableName** (*str(max=512)*) – The full name of the variable to be removed from the subscription.

Returns Returns *NONE* on success.

Return type *DataAccessError*

Subscribe(*subscriptionId: int*, *sampleRate: int*) → *PyPlcnextRsc.Arp.Plc.Gds.Services.DataAccessError*

Subscribes the subscription with the given id.

All previously added variables including in the given subscription will be subscribed. Internally the variables are separated in the respective tasks, a buffer for each task will be created and connected to the task

executed event. At this point the task will copy the selected variable data into the task buffer (excluded subscriptions from kind :py:const:~`PlcnextRsc.Arp.Plc.Gds.Services.SubscriptionKind.DirectRead`). How often the task stores the data to the buffer depends on the task cycle time and the configured subscription sample rate.

Calling this method on a already subscribed subscription has no effect, even if new variables have been added or removed. To make variable modification effective, use *Resubscribe()*. Calling this method while the subscription is in the state Unsubscribed, because *Unsubscribe()* has been called, will only connect the already constructed buffer to the respective tasks and will set the given sampleRate. Compare to the first and initial call of this method, this call cost more less time because the buffer are already created. This also means that variable modification which have been done after the first call of *Subscribe()*, have also no effect. At this point it is also necessary to call *Resubscribe()*.

A subscribed subscription can operates in different sample rates (excluded subscriptions from kind *DirectRead*) which depends on several factors. First, each variable belongs to a program which runs in a task and this task has a cycle rate which determines the sample rate. This means that at the end of each task cycle, all variable data, subscribed and related to this task, will be written to the corresponding buffer. Note that all global variables are assigned to the task 'Globals'. Thats the case if the given sample rate is set to zero, which means the subscription operates in 'real-time', the same sample rate the task is operating in. This is also the fastest possible rate for a subscription. Note that it's possible that one subscription could contain variables from different tasks, which has the consequence that the subscription operates in different rates! If the given sample rate desire to a specific rate, the subscription tries to operate in this rate, for each variable, no matter from which task this variable comes. Potential self defined sample rates for a subscription are the task cycle rate or a multiple of them, otherwise the given rate will rounded down. E.g.:

```
Task A cycle rate = 10ms
Task B cycle rate = 8ms
Subscription given rate = 50ms
Subscription rate for task A = 50ms
Subscription rate for task B = 48ms
```

Special handling for global Varibales: If there isn't a given sample rate by the user (value is zero), the global variables will be recored by default from the 'Globals' task (50ms, configured in the ESM.config). But if there is a given sample rate (value is greater than zero) the global variables will be connected a task which fits the given sample rate. If no task exists with the given sample rate, the fastest available task will be picked and used for downsampling (see above). So it is possible to record data of global variables in the fastest availble interval or an multiple of them.

Parameters

- **subscriptionId** (*int(Uint32)*) – The id of the subscription.
- **sampleRate** (*int(Uint64)*) – The desired sample rate in microseconds.

Returns Returns *NONE* on success

Return type *DataAccessError*

Resubscribe(*subscriptionId: int*, *sampleRate: int*) → *PyPlcnextRsc.Arp.Plc.Gds.Services.DataAccessError*

Resubscribes the subscription with the given id.

Resubscribes the subscription, which will trigger a completely rebuild process of the whole subscription, including previously done variable modification which have been done after the first call of *Subscribe()*. It destroys the internalEnums buffer and subscribes the subscription again (for further information see *Subscribe()*). Note that the subscription is not able to collect data from the variables, while the resubscribe process is in progress. This method has only an effect if the given subscription is currently subscribed, otherwise nothing will happen.

Parameters

- **subscriptionId** (*int(Uint32)*) – The id of the subscription.
- **sampleRate** (*int(Uint64)*) – The desired sample rate in microseconds.

Returns Returns *NONE* on success

Return type *DataAccessError*

Unsubscribe(*subscriptionId: int*) → *PyPlcnextRsc.Arp.Plc.Gds.Services.DataAccessError*

Unsubscribes the subscription with the given id.

Unsubscribes the subscription from all task executed events. The subscription data are still exist and could be get by the respective Read-methods. To subscribe the subscription again, call *Subscribe()*. This method has only an effect if the given subscription is currently subscribed, otherwise nothing will happen.

Parameters **subscriptionId** (*int(Uint32)*) – The id of the subscription.

Returns Returns *NONE* on success

Return type *DataAccessError*

DeleteSubscription(*subscriptionId: int*) → *PyPlcnextRsc.Arp.Plc.Gds.Services.DataAccessError*

Deletes the subscription.

Deletes the subscription with the given id. After that the id is no longer valid and all data, containing in the subscription will be removed.

Parameters **subscriptionId** (*int(Uint32)*) – The id of the subscription.

Returns Returns *NONE* on success

Return type *DataAccessError*

GetVariableInfos(*subscriptionId: int*) → Tuple[Tuple[*PyPlcnextRsc.Arp.Plc.Gds.Services.VariableInfo*], *PyPlcnextRsc.Arp.Plc.Gds.Services.DataAccessError*]

Get the subscribed variable information of the subscription.

The subscription service provides several Read functions

```
Arp.Plc.Gds.Services.ISubscriptionService.ReadValues,
Arp.Plc.Gds.Services.ISubscriptionService.ReadTimeStampedValues
Arp.Plc.Gds.Services.ISubscriptionService.ReadRecords
```

which will return the plain values without any information of type and order. To assign this plain values to the added variables, this function returns the currently subscribed variable information in a array of *VariableInfo* in the same order as the Read functions will do. This order and type information wont change, till *Resubscribe()* was called.

Note: This order does not have to be the same order like the variables has been added to the subscription.

This service function relates to the Read function *ReadValues()*. The provided information contains only information of the added and currently subscribed variables. It doesn't contain information of timestamps. Timestamps could be Read by the function *ReadTimeStampedValues()* and its related information with *GetTimeStampedVariableInfos()*.

```
Example:

    Added Variable from task A: a1, a2
    Added Variable from task B: b1
    Results in:
```

(continues on next page)

(continued from previous page)

```
    VariableInfo[]
    a1
    a2
    b1
```

Parameters **subscriptionId** (*int(Uint32)*) – The id of the subscription.

Returns

tuple with 2 return values :

- A tuple of *VariableInfo*.
- *NONE* on success

Return type tuple[tuple[*VariableInfo*],*DataAccessError*]

GetTimeStampedVariableInfos(*subscriptionId: int*) → Tuple[Tuple[*PyPlcnextRsc.Arp.Plc.Gds.Services.VariableInfo*], *PyPlcnextRsc.Arp.Plc.Gds.Services.DataAccessError*]

Get the subscribed variable information including information of timestamps of the subscription.

TThis service function relates to the Read function *ReadTimeStampedValues()*. The provided information contains information of the added and currently subscribed variables and additionally information about the timestamps. Note that a subscription could contain multiple timestamps, related on the number of used tasks from which the added variables are from. The timestamp is always the first value followed by all to the task related variable information.

```
Example:

    Added Variable from task A: a1, a2
    Added Variable from task B: b1
    Results in:
    VariableInfo[]
    timestamp
    a1
    a2
    timestamp
    b1
```

Each containing timestamp has the variable name timestamp and the data type *Int64* which is provided in *VariableInfo* like each other variable information.

For further information see *GetVariableInfos()*.

Parameters **subscriptionId** (*int(Uint32)*) – The id of the subscription.

Returns

tuple with 2 return values :

- A tuple of *VariableInfo*.
- *NONE* on success

Return type tuple[tuple[*VariableInfo*],*DataAccessError*]

GetRecordInfos(*subscriptionId: int*) → Tuple[Tuple[*PyPlcnextRsc.Arp.Plc.Gds.Services.VariableInfo*], *PyPlcnextRsc.Arp.Plc.Gds.Services.DataAccessError*]

Get the subscribed variable information as a record of the subscription.

> **Warning:** This function is not implemented on the server side, use *GetTimeStampedVariableInfos()* instead.

This service function relates to the Read function *ReadRecords()*. The provided information contains information of the added and currently subscribed variables, its task relation and additionally information about the task related timestamp. The information are provided in an array of array of *VariableInfo*. The first array correspond to the number of different tasks and the second contains the related variable information which are related to the variables of this task and additionally information about the task related timestamp. Each containing timestamp has the variable name timestamp and the data type *Int64* which is provided in *VariableInfo* like each other variable information.

```
Example:

    Added Variable from task A: a1, a2
    Added Variable from task B: b1
    Results in:
    VariableInfo[][]
    VariableInfo[]
    timestamp
    a1
    a2
    VariableInfo[]
    timestamp
    b1
```

For further information see *GetVariableInfos()*.

Parameters **subscriptionId** (*int(Uint32)*) – The id of the subscription.

Returns

tuple with 2 return values :

- A tuple of *VariableInfo*.
- *NONE* on success

Return type tuple[tuple[*VariableInfo*],*DataAccessError*]

ReadValues(*subscriptionId: int*) → Tuple[Tuple[*PyPlcnextRsc.common.objects.rsc_variant.RscVariant*], *PyPlcnextRsc.Arp.Plc.Gds.Services.DataAccessError*]

Read the data from the subscription with the given id.

This service function returns the plain data values from the added and subscribed variables. The data values are returned in a static order and doesn't contain any type information. To figure out which value belongs to the added variable, it is necessary to call the related information function *GetVariableInfos()*. As long as the subscription doesn't resubscribed with *Resubscribe()*, all the information are valid and both, the Read value data and information data, are in a static order. Note that this values doesn't contain timestamps! If the timestamp is needed use the function *ReadTimeStampedValues()* instead.

The Read data may contain null values (*Void*) if the Read call was executed before the tasks initially have written the data.

```
Example:

    Added Variable from task A: a1, a2
    Added Variable from task B: b1
    Results in:
    object[]
    a1
    a2
    b1
```

Parameters **subscriptionId** (*int(Uint32)*) – The id of the subscription.

Returns

tuple with 2 return values :

- Contains the plain values of the given and subscribed variables.
- *NONE* on success

Return type tuple[tuple[*RscVariant*],*DataAccessError*]

ReadTimeStampedValues(*subscriptionId: int*) → Tuple[Tuple[*PyPlcnextRsc.common.objects.rsc_variant.RscVariant*], *PyPlcnextRsc.Arp.Plc.Gds.Services.DataAccessError*]

Read the data including timestamps from the subscription with the given id.

This service function returns the plain data values from the added and subscribed variables including timestamps. The data values are returned in a static order and doesn't contain any type information. To figure out which value belongs to the added variable, it is necessary to call the related information function *GetTimeStampedVariableInfos()*. As long as the subscription doesn't resubscribed with *Resubscribe()*, all the information are valid and both, the Read value data and information data, are in a static order.

The Read data may contain null values (*Void*) if the Read call was executed before the tasks initially have written the data.

```
Example:

    Added Variable from task A: a1, a2
    Added Variable from task B: b1
    Results in:
    object[]
    timestamp task A
    a1
    a2
    timestamp task B
    b1
```

Parameters **subscriptionId** – The id of the subscription.

Returns

tuple with 2 return values :

- Contains the plain values including the timestamps of the given and subscribed variables.
- *NONE* on success

Return type tuple[tuple[*RscVariant*],*DataAccessError*]

ReadRecords(*subscriptionId: int*, *count: int*) → Tuple[Tuple[*PyPlcnextRsc.common.objects.rsc_variant.RscVariant*], *PyPlcnextRsc.Arp.Plc.Gds.Services.DataAccessError*]

Read the data including timestamps from the subscription with the given id separated in task records.

This service function returns the plain data values from the added and subscribed variables including timestamps separated in task records. The data values are returned in a static order and doesn't contain any type information. To figure out which value belongs to the added variable, it is necessary to call the related information function Arp.Plc.Gds.Services.ISubscriptionService.GetRecordInfos. As long as the subscription doesn't resubscribed with Arp.Plc.Gds.Services.ISubscriptionService.Resubscribe, all the information are valid and both, the Read value data and information data, are in a static order.

The number of returned value records depends on the count of tasks, the number of sampled data and the number of the given count parameter.

The structure how the values are returned is strictly defined: The first array (records) contains n arrays (task records) and where n depends on the number of tasks. The array from the second dimension (task records) contains n arrays (record), where n depends on the number of collected data, one data record per task cycle. The array from the third dimension (record) contains the plain values, starting with the timestamp.

The Read data may contain null values (*Void*) if the Read call was executed before the tasks initially have written the data.

```
Example:

    Added Variable from task A: a1, a2
    Added Variable from task B: b1
    task A sampled 2 cycles
    task B sampled 1 cycles
    Results in:
    object[] (records)
    object[] (task A records)
    object[] (record cycle 1)
    timestamp
    a1
    a2
    object[] (record cycle 2)
    timestamp
    a1
    a2
    object[] (task B records)
    object[] (record cycle 1)
    timestamp
    a1
    a2
```

Parameters

- **subscriptionId** (*int(Uint32)*) – The id of the subscription.
- **count** (*int(Uint16)*) – Number of maximum records to be copied per task. If set to zero, all available records will be copied.

Returns

tuple with 2 return values :

- Tuple for the subscribed data records including timestamps.
- *NONE* on success

Return type tuple[tuple[*RscVariant*],*DataAccessError*]

class PyPlcnextRsc.Arp.Plc.Gds.Services.**IForceService**

Service for managing and controlling force variables by the Arp GDS.

Use *IForceService* in order to force and to unforce variables.

AddVariable(*item:* PyPlcnextRsc.Arp.Plc.Gds.Services.ForceItem) → *PyPlcnextRsc.Arp.Plc.Gds.Services.DataAccessError*

Adds a new variable and value for forcing. Enables force mode.

The enabled force mode is signalized by notification and by the activated '*Forcing*'

Parameters **item** – Force item *ForceItem*, which contains the the name of the variable with the full instance path and the force value. The data type of the force value must be equal with the data type of the target variable.

Returns Returns *NONE* on success

Return type *DataAccessError*

RemoveVariable(*variableName: str*)

Resets forced variable. Disables force mode after force list is empty.

Parameters **variableName** (*str(max=512)*) – Instance path of the variable.

GetVariables() → Tuple[*PyPlcnextRsc.Arp.Plc.Gds.Services.ForceItem*]

Gets a list of all forced variables.

Returns Returns a list with all existing *ForceItem* objects.

Return type tuple(*ForceItem*)

Reset()

Resets the force list. Disables force mode.

The disabled force mode is signalized by notification and by the deactivated PlcState.

IsForcable(*variableName: str*) → bool

Tests whether variable is forcable.

The variable has to meet the following requirements to be forcable:

- The kind of variable should be an In- or an Out-port of a program (IEC, C ++, Simulink ...) or a variable that is connected to I/O data.
- The data type of the variable has to be supported.

Returns true if the variable is forcable.

Return type bool

IsActive() → bool

Tests whether force mode is active.

Returns true if the force mode is active.

Return type bool

class PyPlcnextRsc.Arp.Plc.Commons.**DataType**(*value*)

NONE = 0
Unspecified.

Void = 1
Void - Arp C++ empty type

Bit = 2
Bit - Arp C++ data type (1 Byte)

Boolean = 3
Boolean - Arp C++ data type (1 Byte)

UInt8 = 4
UInt8 - Arp C++ data type (1 Byte)

Int8 = 5
Int8 - Arp C++ data type (1 Byte)

Char8 = 6
Char8 - Arp C++ data type (1 Byte)

Char16 = 7
Char16 - Arp C++ data type (2 Byte)

UInt16 = 8
UInt16 - Arp C++ data type (2 Byte)

Int16 = 9
Int16 - Arp C++ data type (2 Byte)

UInt32 = 10
UInt32 - Arp C++ data type (4 Byte)

Int32 = 11
Int32 - Arp C++ data type (4 Byte)

UInt64 = 12
UInt64 - Arp C++ data type (8 Byte)

Int64 = 13
Int64 - Arp C++ data type (8 Byte)

Float32 = 14
Float32 - Arp C++ data type (4 Byte)

Float64 = 15
Float64 - Arp C++ data type (8 Byte)

Primitive = 32
Limit of primitive types

DateTime = 33
C++ DateTime type

IecTime = 34
IEC type: TIME [int32]

IecTime64 = 35
IEC type: LTIME [int64]

IecDate = 36
IEC type: DATE [N/A],Not supported by PCWE.

IecDate64 = 37
IEC type: LDATE [int64]

IecDateTime = 38
IEC type: DATE_AND_TIME, DT [N/A],Not supported by PCWE.

IecDateTime64 = 39
IEC type: LDATE_AND_TIME, LDT [int64]

IecTimeOfDay = 40
IEC type: TIME_OF_DAY, TOD [N/A],Not supported by PCWE.

IecTimeOfDay64 = 41
IEC type: LTIME_OF_DAY, LTOD [int64]

StaticString = 42
Static String type

IecString = 43
Iec String type, only for internal use

ClrString = 44
.NET/C# String type, only for internal use

String = 45
C++ String type, only for internal use

Elementary = 64
Limit of elementary types.

ArrayElement = 65
ArrayOfArray

Struct = 66
Struct

Class = 67
Class

FunctionBlock = 68
Function Block

Subsystem = 69
Subsystem

Program = 70
Program

Component = 71
Component

Library = 72
Library

Complex = 254
Limit of complex types

Pointer = 512
Determines a pointer type.Pointer are declared as *PyPlcnextRsc.Arp.Plc.Commons.DataType.Elementary* kind.

Array = 1024
Determines an array type.Arrays are declared as *PyPlcnextRsc.Arp.Plc.Commons.DataType.Elementary* kind.

Enum = 2048
Determines an Enumeration type.Enums are declared as *PyPlcnextRsc.Arp.Plc.Commons.DataType.Elementary* kind.

Reference = 4096
Determines a C# reference type.Reference are declared as *PyPlcnextRsc.Arp.Plc.Commons.DataType.Elementary* kind.

BaseTypeMask = 255
For removing all flags

2.4.5 Arp.Plc.Domain.Services

class PyPlcnextRsc.Arp.Plc.Domain.Services.**PlcState**(*value*)
An enumeration.

NONE = 0
Not specified.

Ready = 1
The firmware is setup, and the PLC is ready.

Stop = 2
The PLC is loaded and setup but not started yet.

Running = 3
The PLC is started.

Halt = 4
The PLC is halted for debug purpose.

Warning = 64
An unspecified warning occurs.

Error = 128
An unspecified error or exception occurs, and the PLC is in state error.

SuspendedBySwitch = 256
This error bit is set, if it could not be started because the PLC is suspended by the hardware switch (STOP-switch).

FatalError = 512
An unspecified fatal error or exception occurs, and the PLC is in state error.

SuspendedBySystemWatchdog = 1024
This error bit is set, if it could not be started because the PLC is suspended by the system watchdog.

Changing = 65536
The PLC is changing a configuration, this implies, that the state <see cref="Running"/> is set.

Hot = 131072
The PLC is stopped in hot state, that is all data still remains.

Forcing = 262144
The PLC is in force mode. One or more variables are forced by the GDS.

Debugging = 524288
The PLC is in debug mode. One or more breakpoints are set.

Warm = 1048576
The PLC is stopped in warm state, that is the retain data has been restored.

DcgNotPossible = 1073741824
This error bit is set, if the PLC tries to perform a change operation, but it is not possible. This bit is usually combined with the state *Running*

DcgRealTimeViolation = 2147483648
This error bit is set, if the PLC tries to perform a change operation, but it is not possible in real time. # This bit is usually combined with the state *Running*

class PyPlcnextRsc.Arp.Plc.Domain.Services.**PlcStartKind**(*value*)
PLC Start Kind

NONE = 0

Cold = 1
Cold start

Warm = 2
Warm start

Hot = 3
Hot start

class PyPlcnextRsc.Arp.Plc.Domain.Services.**PlcInfoId**(*value*)
all identifiers of Plc informations to be read by IPlcInfoService.

NONE = 0
Not initialized.

ProjectName = 1
The name of the actual project/application.

class PyPlcnextRsc.Arp.Plc.Domain.Services.**IPlcManagerService**
Use this service to control the PLC of the controller.

Load(*Async: bool = False*)
Loads the PLC configuration and setup the PLC.

Parameters **Async** (*bool*) – true, if the operation should be processed asynchronously, otherwise false

Start(*startKind:* PyPlcnextRsc.Arp.Plc.Domain.Services.PlcStartKind, *Async: bool = False*)
Starts the PLC.

StartKind	Description
Cold	A cold start is processed. all data is set to initial values.
Warm	A warm start is processed. all data is set to initial values but retained data is set to the retained values.
Hot	The PLC is just continued without setting or resetting any data.

Parameters

- **startKind** (PlcStartKind) – Determines how the PLC should be started.
- **Async** (*bool*) – true, if the operation should be processed asynchronously, otherwise false

Stop(*Async: bool = False*)
Stops the PLC.

Parameters **Async** (*bool*) – true, if the operation should be processed asynchronously, otherwise false

Reset(*Async: bool = False*)
Resets the PLC and unloads its configuration.

Parameters **Async** (*bool*) – true, if the operation should be processed asynchronously, otherwise false

GetPlcState() → *PyPlcnextRsc.Arp.Plc.Domain.Services.PlcState*
Gets the actual PLC state.

Returns The current PLC state. If the PLC has a warning, the *Warning* Bit is set. If the PLC has an actual error, the *Error* Bit is set.

Return type *PlcState*

class PyPlcnextRsc.Arp.Plc.Domain.Services.**IPlcManagerService2**
The DownloadChange extension of the *IPlcManagerService*.

Change(*Async: bool = False*)
This operation will perform the change of the PLC configuration, which was downloaded before.

Parameters **Async** (*bool*) – true, if the operation should be processed asynchronously, otherwise false.

Restart(*startKind:* PyPlcnextRsc.Arp.Plc.Domain.Services.PlcStartKind, *Async: bool = False*)
Restarts the Plc, that is, it's stopped and started in a single operation.

Parameters

- **startKind** (PlcStartKind) – The start kind *Cold*, PlcStartKind. *Warm* or *Hot*
- **Async** (*bool*) – true, if the operation should be processed asynchronously, otherwise false.

class PyPlcnextRsc.Arp.Plc.Domain.Services.**IPlcInfoService**
Provides informations about the Plc (realtime) project.

GetInfo(*identifier:* PyPlcnextRsc.Arp.Plc.Domain.Services.PlcInfoId) → *PyPlcnextRsc.common.objects.rsc_variant.RscVariant*
Gets the specified info from the Plc project.

Parameters **identifier** (PlcInfoId) – The identifier of the info to Read.

Returns The requested info or null if the identifier is unknown.

Return type *RscVariant*

GetInfos(*identifiers: Sequence[*PyPlcnextRsc.Arp.Plc.Domain.Services.PlcInfoId*]*) → Tuple[*PyPlcnextRsc.common.objects.rsc_variant.RscVariant*]
Gets the specified infos from the Plc project.

Parameters **identifiers** (*Sequence[*PlcInfoId*]*) – The identifiers of the infos to Read.

Returns The requested infos as tuple. If an identifiers is unknown, the according array element is null.

Return type typle[*RscVariant*]

2.4.6 Arp.Device.Interface.Services

class PyPlcnextRsc.Arp.Device.Interface.Services.**AccessErrorCode**(*value*)

Enumeration for error codes returned from Device.Interface.Services.

NONE = 0
Success

UnknownError = 1
Unkown Error

UnknownSetting = 2
Unknown setting id

AuthorizationFailure = 3
Authorization failure

IncompatibleType = 4
The type of the new value is wrong

InvalidFormat = 5
The format of the new value is wrong

InvalidParameter = 6
A parameter is invalid

OutOfRange = 7
A value is out of range

class PyPlcnextRsc.Arp.Device.Interface.Services.**DeviceSettingResult**(*ErrorCode:* PyPlcnextRsc.Arp.Device.Interface.Services.AccessEr *Value:* PyPlcnextRsc.common.objects.rsc_variant.RscVariant

Container for a combination of an error code and a value which is a result for a Read operation to a single setting.

The structure is used as result by *PyPlcnextRsc.Arp.Device.Interface.Services.IDeviceSettingsService*. ReadValue A sequence of instances of this structure is used as result by *ReadValues()*.

ErrorCode: *PyPlcnextRsc.Arp.Device.Interface.Services.AccessErrorCode*
An error code which describes the reason why a setting could not be Read with The error codes can be interpreted with the help of the enumeration *PyPlcnextRsc.Arp.Device.Interface.Services.AccessErrorCode*.

Value: *PyPlcnextRsc.common.objects.rsc_variant.RscVariant*
The value for the requested setting which can be *PyPlcnextRsc.common.tag_type.RscType.Void* and otherwise is of the simple type the setting has. Please note that this property is *PyPlcnextRsc.common.tag_type.RscType.Void* whenever the member *ErrorCode* indicates an error which means it is not *PyPlcnextRsc.Arp.Device.Interface.Services.AccessErrorCode.NONE*.

class PyPlcnextRsc.Arp.Device.Interface.Services.**DeviceSettingItem**(*Setting: str*, *Value:* PyPlcnextRsc.common.objects.rsc_variant.RscVariant)

Container for a (relative) setting identifier with its value.

The structure is designed for efficient interpretation on the receiving side which is the Remoting server. The structure is used by "*PyPlcnextRsc.Arp.Device.Interface.Services.IDeviceSettingsService.WriteValue()*" A sequence of instances of this structure is intended to be passed to the method *PyPlcnextRsc.Arp.Device.Interface.Services.IDeviceSettingsService.WriteValues()*

Setting: str
Tokens which describe the (relative) path to the setting combined with the value. If this instance of the structure TokensAndValue is part of a sequence (an array) then the preceeding instance's tokens already provide a context at which the tokens here start with their description.

Value: *PyPlcnextRsc.common.objects.rsc_variant.RscVariant*
The value of the setting which is identified by the (relative) path which is described by the tokens. The type of the value must match the value which is defined for the particular setting.

class PyPlcnextRsc.Arp.Device.Interface.Services.**IDeviceInfoService**
Use this service to Read Device information.

GetItem(*identifier: str*) → *PyPlcnextRsc.common.objects.rsc_variant.RscVariant*
Read a single information

Returns value as RscVariant on success, *PyPlcnextRsc.common.tag_type.RscType.Void* on error

Return type *RscVariant*

GetItems(*identifiers: Sequence[str]*) → Tuple[*PyPlcnextRsc.common.objects.rsc_variant.RscVariant*]
Read a list of information

Parameters **identifiers** – Sequence of String for select the items

Returns values as RscVariant on success, *PyPlcnextRsc.common.tag_type.RscType.Void* on error

Return type tuple[*RscVariant*]

class PyPlcnextRsc.Arp.Device.Interface.Services.**IDeviceStatusService**
Use this service to Read Device states.

GetItem(*identifier: str*) → *PyPlcnextRsc.common.objects.rsc_variant.RscVariant*
Read a single state

Returns value as RscVariant on success, *PyPlcnextRsc.common.tag_type.RscType.Void* on error

Return type *RscVariant*

GetItems(*identifiers: Sequence[str]*) → Tuple[*PyPlcnextRsc.common.objects.rsc_variant.RscVariant*]
Read a list of state

Parameters **identifiers** (*Sequence[str(max=512)]*) – Array of String for select the items

Returns values as RscVariant on success, *PyPlcnextRsc.common.tag_type.RscType.Void* on error

Return type tuple[*RscVariant*]

class PyPlcnextRsc.Arp.Device.Interface.Services.**IDeviceSettingsService**
Use this service to Read and Write Device settings.

ReadValue(*setting: str*) → *PyPlcnextRsc.Arp.Device.Interface.Services.DeviceSettingResult*
Read a single setting

Parameters **setting** (*str(max=512)*) – String for select the item

Returns result as *DeviceSettingResult*

Return type *DeviceSettingResult*

ReadValues(*settings: Sequence[str]*) → Tuple[*PyPlcnextRsc.Arp.Device.Interface.Services.DeviceSettingResult*]

Read a list of settings

Parameters **settings** (*Sequence[str(max=512)]*) – Sequence of string for select the items

Returns result as array of *DeviceSettingResult*

Return type tuple[*DeviceSettingItem*]

WriteValue(*settingItem:* PyPlcnextRsc.Arp.Device.Interface.Services.DeviceSettingItem) → *PyPlcnextRsc.Arp.Device.Interface.Services.AccessErrorCode*

Write a single setting

Parameters **settingItem** (DeviceSettingItem) – *DeviceSettingItem* with string for select the item and the new value

Returns result as *AccessErrorCode*

Return type *AccessErrorCode*

WriteValues(*settingItems: Sequence[*PyPlcnextRsc.Arp.Device.Interface.Services.DeviceSettingItem*]*) → Tuple[*PyPlcnextRsc.Arp.Device.Interface.Services.AccessErrorCode*]

Write a list of settings

Parameters **settingItems** (*Sequence[*DeviceSettingItem*]*) – Sequence of *DeviceSettingItem* for set the items to new values

Returns result as tuple of *AccessErrorCode*

Return type tuple[*AccessErrorCode*]

class PyPlcnextRsc.Arp.Device.Interface.Services.**IDeviceControlService**

Use this service to control the Device.

RestartDevice()

Reboot the Device

ResetToFactoryDefaults(*resetType: int*) → *PyPlcnextRsc.Arp.Device.Interface.Services.AccessErrorCode*

Reset Device configuration

After successful start the PLC will stop and than reboot. While the reboot the requested defaults will be set.

Reset type	Description
1	Reset Device configuration to factory default.
2	Downgrade FW to factory version and reset configuration.

Parameters **resetType** (*int(Uint16)*) – see the table above.

Returns Result of start execute

Return type *AccessErrorCode*

StartFirmwareUpdate(*updateType: int*) → *PyPlcnextRsc.Arp.Device.Interface.Services.AccessErrorCode*

Start FW update

Before you can start an update a raucb-container must be copied to path "/opt/plcnext". After successfull start an update the PLC will stop, execute the update and than reboot.

Parameters **updateType** (*int(Uint16)*) – Reserved for extensions, must be 0 in this version.

Returns Result of start execute

Return type *AccessErrorCode*

2.4.7 Arp.System.Security.Services

class PyPlcnextRsc.Arp.System.Security.Services.**FileSystemError**(*value*)
This enum is used by several file operations.

NONE = 0
Success

Unknown = 1
The error is not listed in this enumeration.

InvalidPath = 2
The path is invalid.

NotExist = 3
The path does not exist.

AlreadyExists = 4
The path already exists.

AccessDenied = 5
The file is already in use.

OutOfSpace = 6
There is not enough space on the Device left.

class PyPlcnextRsc.Arp.System.Security.Services.**AuthenticationError**(*value*)
Defines values indicating success or failure of an attempt to create a security session.

NONE = 0
A new security session has been established successfully.

InvalidCredentials = -1
Authentication failed because name and / or password is not correct.

PeraltyDelayActive = -2
Authentication failed because a penalty delay is currently active.

DuplicateSession = -3
For this combination of user and rscClient there already exists a session.

SessionLimitReached = -4
The remote side's capacity does no allow to open more security sessions within this rscClient.

TryAgainLater = -5
The system is temporarily unable to answer authentication requests.

PasswordMustBeChanged = -6
The password is correct but it must be changed before a session is created.

class PyPlcnextRsc.Arp.System.Security.Services.**PasswordChangeError**(*value*)
Defines values indicating success or failure of an attempt to create a security session.

NONE = 0
The password has been changed or set as desiged.

InvalidCredentials = -1
Authentication (during password change) failed because name and / or the (old) password is not correct.

PenaltyDelayActive = -2
Authentication (during password change) failed because a penalty delay is currently active in example due to too many failed authentication attempts.

NotSupportedForThisUser = -3
The (sub)system in use for authentication of the user does not support changing or (re)setting a password.

InsufficientNewPassword = -4
The password does not comply with security policies.

TryAgainLater = -5
The system is temporarily unable to answer authentication requests.

class PyPlcnextRsc.Arp.System.Security.Services.**IPasswordAuthenticationService**
This service allows a Remoting client to authenticate a user to the gateway (device) and by this start a security session on behalf of her or him. With PLCnext Technology authentication happens with usernames. But other kind of devices might not identify individual users but authenticate roles. Thus the name parameters given here in this service interface refer to role names or other kinds of identities then.

createSession(*username: str*, *password: str*) → Tuple[PyPlcnextRsc.common.objects.SecurityToken, int, *PyPlcnextRsc.Arp.System.Security.Services.AuthenticationError*]
Starts a security context for a particular user or role known at the device. Every successful call (see return value) to this method creates a security session for a particular user (or role) at the RSC-gateway (device). Within the same rscClient a client can create several sessions. Sessions are identified by the securityToken provided by this method. But please check the T:Ade.CommonRemoting.IRemotingAdapter whether and how it allows to change the securityToken to use within the rscClient. First versions of this interface requires synchronization, so switching security sessions on a per call basis did not make sense for this because it is rather expensive. The maximum number of concurrent sessions is limited by the gatway's capacitiy (respectively the device's capacity) and therefore clients shall terminate sessions which they do not need any more, *PyPlcnextRsc.Arp.System.Security.Services.IPasswordAuthenticationService.closeSession()*. Sessions are automatically terminated whenever the corresponding Remoting rscClient is closed. Sessions cannot be used across channels.

Parameters

- **username** – The name of the role or user or other identity in question,With PLCnext Technology the name has to be a username, not a role name.
- **password** – The password which is the proof that the client is allowed to act as the user (or role) specified with username.

  Please note that the password is sensitive data. The caller must care for keeping it secret. If the method implementation creates copies then it ensures they are erased from memory (i.e. overwritten with zeroes) before destruction and that each copy is destructed at latest when the corresponding security context is closed.

Returns

tuple with 3 return values :

0. **securityToken - A value identifying the security session.** The output value is only valid in case that the method has the return value *PyPlcnextRsc.Arp.System.Security.Services.AuthenticationError.NONE* to indicate success. Then and only then a security session has been created which shall be closed by the client with a call to *PyPlcnextRsc.Arp.System.Security.Services.IPasswordAuthenticationService.closeSession()* once it is no longer needed. All security sessions which have been opened by the client during a Remoting connection are automatically closed whenever the Remoting connection is terminated. The value is intended to be passed to the Remoting sink of the communication rscClient, in example with every method call. Session context handles shall not be interpreted by the client.

The client cannot derive any further meaning from the value.One exception: The value 0 (zero) is never returned by this upon success. The value zero always identifies the default security session which is used when no particular security session has been established yet,in example right after the Remoting rscClient was established. Thus for safety reasons the out parameter securityToken is set to zero whenever this method indicates an error.

1. **penaltyDelayMillis - A timespan expressed in milliseconds for which further authentication attempts are r**
   The value is always 0 unless the method returns *PyPlcnextRsc.Arp.System.Security.Services.PasswordChangeError.InvalidCredentials*. Then it is a positive value. A penalty delay is imposed to slow down brute force attacks. The penalty may be valid for a particular user attempting to change a password or to authenticate via a particular access path (TCP port, service, ...) or it may be valid for a group or even all users. The timespan is just an indication that subsequent authentication attempts may be rejected during this timespan. Please note that the accuracy of this value is questionable due to the latency which is introduced with its transmission.
2. **Upon success *PyPlcnextRsc.Arp.System.Security.Services.AuthenticationError.NONE* is returne**
   Every other value indicates a failure.

Return type tuple[SecurityToken,int,bool]

closeSession(*securityToken: PyPlcnextRsc.common.objects.SecurityToken*)
Terminates a security session which was started at the gateway (device) by a former call to *createSession*

Parameters **securityToken** (*SecurityToken*) – Security token as formerly returned by a successful session creation. It identifies the session which shall be closed.

class PyPlcnextRsc.Arp.System.Security.Services.**IPasswordConfigurationService2**
This service allows to maintain passwords for existing users (password based authentication).

changePassword(*username: str*, *oldPassword: str*, *newPassword: str*) → Tuple[int, *PyPlcnextRsc.Arp.System.Security.Services.PasswordChangeError*]
This operation changes a password for a user.

The method can even be called without prior authentication because it verifies the old password before it sets the new password. Support for calls without authentication is required to allow users to change their password even if the password has been locked in example because its usage period has expired. Systems which enforce password changes before and after password expiration then can allow anybody to call this method to unlock their password prior to the next authentication.

Parameters

- **username** (*str(max=128)*) – The new who's password to change.
- **oldPassword** – The old password for verification.
- **newPassword** – The new password to set.

Returns

tuple with 2 return values :

0. **penaltyDelayMillis - A timespan expressed in milliseconds for which further authentication attempts are r**
   The value is always 0 unless the method returns *PyPlcnextRsc.Arp.System.Security.Services.PasswordChangeError.InvalidCredentials*. Then it is a positive value. A penalty delay is imposed to slow down brute force attacks. The penalty may be valid for a particular user attempting to change a password or to authenticate via a particular access path (TCP port, service, ...) or it may be valid for a group or even all users. The timespan is just an indication that subsequent authentication attempts may be rejected during this timespan.

1. **Upon success** *PyPlcnextRsc.Arp.System.Security.Services.PasswordChangeError.NONE* **is returne**
   Every other value indicates a failure.

Return type tuple[int,*PasswordChangeError*]

setPassword(*username: str*, *newPassword: str*) → *PyPlcnextRsc.Arp.System.Security.Services.PasswordChangeError*

This operation overrides the password for a user with administrative power. This operation is an administration function because it allows to change a password for a user without knowing it. Thus appropriate access control lists need to be in place and the method must only be callable for users with administrative rights.

:param username:The name of the user for whom to override the password. :type username: str(max=64) :param newPassword: The new password to set. :type newPassword: str(max=128)

Returns Upon success :py:const:'PyPlcnextRsc.Arp.System.Security.Services.PasswordChangeError.NONE' is returned. Every other value indicates a failure. The :py:const:'PyPlcnextRsc.Arp.System.Security.Services.PasswordChangeError.PenaltyDelayActive` is never returned by this method.

Return type *PasswordChangeError*

class PyPlcnextRsc.Arp.System.Security.Services.**ISecureDeviceInfoService**

This service provides information about the access control which applies to the device without relation to any session or former authorization.

getSystemUseNotification() → Tuple[*PyPlcnextRsc.common.objects.rsc_stream.RscStream*, *PyPlcnextRsc.Arp.System.Commons.Services.Io.FileSystemError*]

Provides a message for display to users before accessing the device that access restrictions apply and what the consequence of ignoring them is. The system use notification conventionally is needed due to legal issues. If an attacker has not been warned that he tries to abuse an access restricted device then much lower legal consequences he has to expect. So the message is intended for warning about entering an access restricted area (device) and which consequences it has if the area is entered without authorization. The message shall be displayed by clients to the user at the time the user establishes a session to the device.

Returns

tuple with 2 return values :

0. **The system use notification to display to users before they start a session to access the device.**
   The message is provided as array of strings which need to be concatenated for display. The message is configured by the device owner. Which language it is depends on the owner. It is likely that the message contains the notification in several languages.

1. An error indication whether the message could be provided correctly.

Return type tuple[*RscStream*,*FileSystemError*]

class PyPlcnextRsc.Arp.System.Security.Services.**ISecuritySessionInfoService2**

This service provides information about the access crontol which currently applies to the session and which was established for example with *IPasswordAuthenticationService*.

getRoleNames() → Tuple[str]

Provides the set of roles which determines the kind of access which is granted. It is intended for use within error messages which explain why a particular access to the gateway (device) is currently not allowed. The role names may be cached for the lifetime of a security session. But a caller must not assume that with the next authentication attempt for the same user the roles will stay the same.

Returns The names of the roles as they are associated with the current session at the gateway site (device site).

Return type tuple[str]

isServiceCallAllowed(*provider: str*, *service: str*, *method: str*) → bool

Allows to determine which methods of which services can currently be called by this client within the current security session. Please note that the access which is granted depends on the current security context which applies to the session at the server site (device site). This is changed if the client authenticates to the server (another time) during the Remoting connection for example by using the *PyPlcnextRsc.Arp.System.Security.Services.IPasswordAuthenticationService*. Clients may cache and re-use the infomation as long as a correspondig security session is valid.

Parameters

- **provider** (*str(max=128)*) – Must be the name which identifies the provider of the service. The same service may be implemented by different providers.
- **service** (*str(max=128)*) – Must be the name which identifies the service. Conventionally this is the fully qualified name of the C# interface which defines the service.
- **method** (*str(max=128)*) – Must be the number of the method.

Returns A value which indicates whether the service method is allowed to be invoked or not.

Return type bool

isUserAuthenticationRequired() → bool

Defines whether user authentication is required. At the device user authentication can be disabled with the effect that anything is allowed.

Returns If user authentication is required true is returned and then without authentication nothing except authentication is allowed.

Return type bool

isServiceCallAllowedOld(*provider: str*, *service: str*, *method: int*) → bool

Allows to determine which methods of which services can currently be called by this client within the current security session. Please note that the access which is granted depends on the current security context which applies to the session at the server site (device site). This is changed if the client authenticates to the server (another time) during the Remoting connection for example by using the *IPasswordAuthenticationService*. Clients may cache and re-use the infomation as long as a correspondig security session is valid.

Parameters

- **provider** (*str(max=128)*) – Must be the name which identifies the provider of the service. The same service may be implemented by different providers.
- **service** (*str(max=128)*) – Must be the name which identifies the service. Conventionally this is the fully qualified name of the C# interface which defines the service.
- **method** (*int*) – Must be the number of the method.

Returns A value which indicates whether the service method is allowed to be invoked or not.

Return type bool

CHAPTER

THREE

SERVICE SUMMARY

There are still some other RSC interfaces this package not covered currently, or some new services just released in the latest firmware. If you need any other interface include, see the source code of services that already written in this package as template demo and refer to the interface prototype in CppSDK , then you can write interface for this package by yourself.

Also, you can send author an Email for adding those service interfaces you want. songyantao@phoenixcontact.com.cn

Here are the Services already implemented in this package:

PyPlcnextRsc.Arp.Services.NotificationLogger.Services.INotificationLoggerService()	The NotificationLogger stores Notifications and provides an interface to retrieve them.
PyPlcnextRsc.Arp.Services.DataLogger.Services.IDataLoggerService2()	The DataLogger provides an interface to log and store variables during firmware runtime.
PyPlcnextRsc.Arp.System.Commons.Services.Io.IDirectoryService()	Provides common file directory operations.
PyPlcnextRsc.Arp.System.Commons.Services.Io.IFileService()	Provides common file operations for reading and writing files as well as deleting/moving/copying files on the Device.
PyPlcnextRsc.Arp.System.Commons.Services.Io.IFileSystemInfoService()	A generic service to retrieve file system infos.
PyPlcnextRsc.Arp.Plc.Gds.Services.IDataAccessService()	Services for the direct data access.
PyPlcnextRsc.Arp.Plc.Gds.Services.ISubscriptionService()	Services for the subscription.
PyPlcnextRsc.Arp.Plc.Gds.Services.IForceService()	Service for managing and controlling force variables by the Arp GDS.
PyPlcnextRsc.Arp.Plc.Domain.Services.IPlcManagerService()	Use this service to control the PLC of the controller.
PyPlcnextRsc.Arp.Plc.Domain.Services.IPlcManagerService2()	The DownloadChange extension of the *IPlcManagerService*.
PyPlcnextRsc.Arp.Plc.Domain.Services.IPlcInfoService()	Provides informations about the Plc (realtime) project.
PyPlcnextRsc.Arp.Device.Interface.Services.IDeviceInfoService()	Use this service to Read Device information.
PyPlcnextRsc.Arp.Device.Interface.Services.IDeviceStatusService()	Use this service to Read Device states.
PyPlcnextRsc.Arp.Device.Interface.Services.IDeviceSettingsService()	Use this service to Read and Write Device settings.

continues on next page

Table 1 – continued from previous page

PyPlcnextRsc.Arp.Device.Interface.Services.IDeviceControlService()	Use this service to control the Device.
PyPlcnextRsc.Arp.System.Security.Services.IPasswordAuthenticationService()	This service allows a Remoting client to authenticate a user to the gateway (device) and by this start a security session on behalf of her or him.
PyPlcnextRsc.Arp.System.Security.Services.IPasswordConfigurationService2()	This service allows to maintain passwords for existing users (password based authentication).
PyPlcnextRsc.Arp.System.Security.Services.ISecureDeviceInfoService()	This service provides information about the access control which applies to the device without relation to any session or former authorization.
PyPlcnextRsc.Arp.System.Security.Services.ISecuritySessionInfoService2()	This service provides information about the access crontol which currently applies to the session and which was established for example with *IPasswordAuthenticationService*.

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W