PyTango:
Python bindings for Tango
v.3.0.0

M. Ounsy, A. Buteau, V.Forchì, E. Taurel

July 26, 2006

History of modifications

Date	Revision	Description	Author
18/07/03	1.0	Initial Version	M. Ounsy
6/10/03	2.0	Extension of the "Getting Started" paragraph	A. Buteau/M. Ounsy
14/10/03	3.0	Added Exception Handling paragraph	M. Ounsy
13/06/05	4.0	Ported to LaTeX, added events, AttributeProxy and ApiUtil	V. Forchì
13/06/05	4.1	Fixed bug with python 2.4 and state events new Database constructor	V. Forchì
15/01/06	5.0	Added Device Server classes	E.Taurel

Contents

1 Introduction

The python Tango binding is made accessible using the module **PyTango**. The PyTango python module implements the Python Tango Device and Database API mapping but also allow to write Tango device server using Python. It then allows access from Python environment to the Tango high level C++ classes and structures (see "The TANGO Control system manual" for complete reference). If you want to write Tango device server in Python using this **PyTango** module, you need Tango C++ library release 5.5 or above. If you simply need to write Python script which act as Tango client only, older releases of tango can be used.

These Tango high level C++ classes and structures are exported to Python using the Boost.python library (see http://boost.sourceforge.net). Details on how to install the boost library and generate the python PyTango module from source code are available in the Readme files of this packages.

2 Getting started

2.1 Binding Installation

To use the Python Binding, two dlls *PyTango.pyd* and *boost_python.dll* (on Windows) or *PyTango.so* and *libboost_python.so* (on Linux) have to be generated. See the package Readme files to find out how these libraries can be generated.

2.2 A quick tour of client binding through real examples

To access Tango devices in your python script, the following line must be in the header of your script file (it's purpose is to import the Tango binding in the interpreter)

```
from PyTango import *
```

You are now ready to execute your first PyTango script!

Example 2.1 *Test the connection to the Device and get it's current state. See test_ping_state.py.*

```
from PyTango import *
import sys
import time

# Protect the script from Exceptions
try:
        # Get proxy on the tangotest1 device
        print "Getting DeviceProxy "
        tangotest = DeviceProxy("tango/tangotest/1")

        # First use the classical command_inout way to execute the DevString command
        # (DevString in this case is a command of the TangoTest device)

        result= tangotest.command_inout("DevString", "First hello to device")
        print "Result of execution of DevString command=", result

        # the same with a Device specific command
        result= tangotest.DevString("Second Hello to device")
        print "Result of execution of DevString command=", result

        # Please note that argin argument type is automagically managed by python
        result= tangotest.DevULong(12456)
        print "Result of execution of Status command=", result

# Catch Tango and Systems  Exceptions
except:
        print "Failed with exception !"
        print sys.exc_info()[0]
```

Example 2.2 *Execute commands with scalar arguments on a Device.*
See test_simple_commands.py. As you can see in the following example, when scalar types are used, the Tango binding automagically manages the data types, and writing scripts is quite easy.

```
tangotest = DeviceProxy("tango/tangotest/1")

# First use the classical command_inout way to execute the DevString command
# (DevString in this case is a command of the TangoTest device)

result= tangotest.command_inout("DevString", "First hello to device")
print "Result of execution of DevString command=", result

# the same with a Device specific command
result= tangotest.DevString("Second Hello to device")
print "Result of execution of DevString command=", result

# Please note that argin argument type is automagically managed by python
result= tangotest.DevULong(12456)
print "Result of execution of Status command=", result
```

Example 2.3 *Execute commands with more complex types.*
See test_complex_commands.py. In this case you have to use put your arguments data in the correct python structures.

```
print "Getting DeviceProxy "
tango_test = DeviceProxy("tango/tangotest/1")
# The input argument is a DevVarLongStringArray
# so create the argin variable containing
# an array of longs and an array of strings
argin = ([1,2,3], ["Hello", "TangoTest device"])

result= tango_test.DevVarLongArray(argin)
print "Result of execution of DevVarLongArray command=", result
```

Example 2.4 *Reading and writing attributes: test_read_write_attributes.py*

```
#Read a scalar attribute
scalar=tangotest.read_attribute("long_scalar")
print "attribute value", scalar.value
#Read a spectrum attribute
spectrum=tangotest.read_attribute("double_spectrum")
print "attribute value", spectrum.value

# Write a scalar attribute so use the scalar structure
print "Writing attributes"

# Write a scalar attribute so use the scalar structure
scalar.value = 18
print "attribute scalar  ", scalar
print "Writing scalar attributes"
tangotest.write_attribute(scalar)

# Write a scalar attribute so use the scalar structure
spectrum.value = [1.2,3.2,12.3]
print "attribute spectrum ", spectrum
print "Writing spectrum attributes"
tangotest.write_attribute(spectrum)
```

Example 2.5 *Defining devices in the Tango DataBase. See test_device_creation.py.*

```
#  A reference on the DataBase
db = Database()

# The 3 devices name we want to create
# Note: these 3 devices will be served by the same DServer
new_device_name1="px1/tdl/mouse1"
new_device_name2="px1/tdl/mouse2"
new_device_name3="px1/tdl/mouse3"

# Define the Tango Class served by this  DServer
new_device_info_mouse = DbDevInfo()
new_device_info_mouse._class = "Mouse"
new_device_info_mouse.server = "ds_Mouse/server_mouse"

# add the first device
print "Creation Device:" , new_device_name1
new_device_info_mouse.name = new_device_name1
db.add_device(new_device_info_mouse)

# add the next device
print "Creation Device:" , new_device_name2
new_device_info_mouse.name = new_device_name2
db.add_device(new_device_info_mouse)
# add the third device
print "Creation Device:" , new_device_name3
new_device_info_mouse.name = new_device_name3
db.add_device(new_device_info_mouse)
```

Example 2.6 *Setting up Device properties. See test_ device_ properties.py.*

Example 2.7 *A more complex example using python subtilities. The following python script example (containing some functions and instructions manipulating a Galil motor axis device server) gives an idea of how the Tango API should be accessed from Python*

```
# connecting to the motor axis device
axis1 = DeviceProxy ("microxas/motorisation/galilbox")
# Getting Device Properties
property_names = ["AxisBoxAttachement",
                  "AxisEncoderType",
                  "AxisNumber",
                  "CurrentAcceleration",
                  "CurrentAccuracy",
                  "CurrentBacklash",
                  "CurrentDeceleration",
                  "CurrentDirection",
                  "CurrentMotionAccuracy",
                  "CurrentOvershoot",
                  "CurrentRetry",
                  "CurrentScale",
                  "CurrentSpeed",
                  "CurrentVelocity",
                  "EncoderMotorRatio",
                  "logging_level",
                  "logging_target",
                  "UserEncoderRatio",
                  "UserOffset"]
axis_properties = axis1.get_property(property_names)
for prop in axis_properties.keys():
print "%s: %s" % (prop,axis_properties[prop][0])
# Changing Properties
```

```
axis_properties["AxisBoxAttachement"] = ["microxas/motorisation/galilbox"]
axis_properties["AxisEncoderType"] = ["1"]
axis_properties["AxisNumber"] = ["6"]
axis1.put_property(axis_properties)
# Reading attributes and storing them in a python dictionary
att_dict = {}
att_list = axis.get_attribute_list()
for att in att_list:
    att_val = axis.read_attribute(att)
    print "%s: %s" % (att,att_val.value)
    att_dict[att] = att_val
# Changing some  attribute values
attributes["AxisBackslash"].value = 0.5
axis1.write_attribute(attributes["AxisBackslash"])
attributes["AxisDirection"].value = 1.0
axis1.write_attribute(attributes["AxisDirection"])
attributes["AxisVelocity"].value = 1000.0
axis1.write_attribute(attributes["AxisVelocity"])
attributes["AxisOvershoot"].value = 500.0
axis1.write_attribute(attributes["AxisOvershoot"])
# Testing some device commands
pos1=axis1.read_attribute("AxisCurrentPosition")
axis1.command_inout("AxisBackward")
while pos1.value > 1000.0:
     pos1=axis1.read_attribute("AxisCurrentPosition")
     print "position axis 1 = ",pos1.value
axis1.command_inout("AxisStop")
```

2.3 A quick tour of Tango device server binding through an example

To write a tango device server in python, you need to import two modules in your script which are:

1. The **PyTango** module
2. The python **sys** module provided in the classical python distribution

The following in the python script for a Tango device server with two commands and two attributes. The commands are:

1. IOLOng which receives a Tango Long and return it multiply by 2. This command is allowed only if the device is in the ON state.
2. IOStringArray which receives an array of Tango strings and which returns it but in the reverse order. This command is only allowed if the device is in the ON state.

The attributes are:

1. Long_attr wich is a Tango long attribute, Scalar and Read only with a minimum alarm set to 1000 and a maximum alarm set to 1500
2. Short_attr_rw which is a Tango short attribute, Scalar and Read/Write

The following code is the complete device server code.

```
    #
    import PyTango
    import sys
    #
    #
    class PyDsExp(PyTango.Device_3Impl):
       def __init__(self,cl,name):
          PyTango.Device_3Impl.__init__(self,cl,name)
```

```
      print 'In PyDsExp __init__'
      PyDsExp.init_device(self)
#
   def init_device(self):
      print 'In Python init_device method'
      self.set_state(PyTango.DevState.ON)
      self.attr_short_rw = 66
      self.attr_long = 1246
#------------------------------------------------------------------
   def delete_device(self):
      print "[Device delete_device method] for device",self.get_name()
#------------------------------------------------------------------
   def is_IOLong_allowed(self):
      if (self.get_state() == PyTango.DevState.ON):
        return True
      else:
        return False
#
   def IOLong(self,in_data):
      print "[IOLong::execute] received number",in_data
      in_data = in_data * 2;
      print "[IOLong::execute] return number",in_data
      return in_data;
#------------------------------------------------------------------
   def is_IOStringArray_allowed(self):
      if (self.get_state() == PyTango.DevState.ON):
         return True
      else:
         return False
#
   def IOStringArray(self,in_data):
      l = range(len(in_data)-1,-1,-1);
      out_index=0
      out_data=[]
      for i in l:
         print "[IOStringArray::execute] received String",in_data[out_index]
         out_data.append(in_data[i])
         print "[IOStringArray::execute] return String",out_data[out_index]
         out_index = out_index+1
      self.y = out_data
      return out_data
#------------------------------------------------------------------
# ATTRIBUTES
#------------------------------------------------------------------
   def read_attr_hardware(self,data):
      print 'In read_attr_hardware'
#------------------------------------------------------------------
   def read_Long_attr(self,the_att):
      print "[PyDsExp::read_attr] attribute name Long_attr"
      PyTango.set_attribute_value(the_att,self.attr_long)
#------------------------------------------------------------------
   def read_Short_attr_rw(self,the_att):
      print "[PyDsExp::read_attr] attribute name Short_attr_rw"
      PyTango.set_attribute_value(the_att,self.attr_short_rw)
#------------------------------------------------------------------
   def write_Short_attr_rw(self,the_att):
      print "In write_Short_attr_rw for attribute ",the_att.get_name()
      data=[]
      PyTango.get_write_value(the_att,data)
```

```
      self.attr_short_rw = data[0]
#
#
#
#
class PyDsExpClass(PyTango.PyDeviceClass):
   def __init__(self,name):
      PyTango.PyDeviceClass.__init__(self,name)
      self.set_type("TestDevice")
      print 'In PyDsExpClass __init__'

   cmd_list = {'IOLong':[[PyTango.ArgType.DevLong,"Number"],[PyTango.ArgType.DevLong,"Number * 2"]
'IOStringArray':[[PyTango.ArgType.DevVarStringArray,"Array of string"],[PyTango.ArgType.DevVarStri
}

   attr_list = {'Long_attr':[[PyTango.ArgType.DevLong,PyTango.AttrDataFormat.SCALAR,PyTango.AttrWr
{'min alarm':1000,'max alarm':1500}],
'Short_attr_rw':[[PyTango.ArgType.DevShort,PyTango.AttrDataFormat.SCALAR,PyTango.AttrWriteType.REA
}
#
#
#
if __name__ == '__main__':
   try:
     py = PyTango.PyUtil(sys.argv)
     py.add_TgClass(PyDsExpClass,PyDsExp,'PyDsExp')

     U = PyTango.Util.instance()
     U.server_init()
     U.server_run()

   except PyTango.DevFailed,e:
     print '-------> Received a DevFailed exception:',e
   except Exception,e:
     print '-------> An unforeseen exception occured....',e
```

3 The Tango Device Python API

The PyTango allows access from Python environment to the following Tango high level C++ Device classes and structures:

- DeviceInfo
- DbDevImportInfo
- CommandInfo
- TimeVal
- DeviceDataHistory
- AttributeInfo
- AttributeValue
- DeviceAttributeHistory
- DbDevInfo
- DbDevExportInfo
- Database

Additionally, Tango enumerated types are mapped
to named python constants as follows,

DevSource enumeration values are mapped to the following integer constants:

- DevSource.DEV
- DevSource.CACHE
- DevSource.CACHE_DEV

DispLevel enumeration values are mapped to the following integer constants:

- DispLevel.OPERATOR
- DevSource.EXPERT

AttrWriteType enumeration values are mapped to the following integer constants:

- AttrWriteType.READ
- AttrWriteType.READ_WITH_WRITE
- AttrWriteType.WRITE
- AttrWriteType.READ_WRITE

AttrDataFormat enumeration values are mapped to the following integer constants:

- AttrDataFormat.SCALAR
- AttrDataFormat.SPECTRUM
- AttrDataFormat.IMAGE

AttrQuality enumeration values are mapped to the following integer constants:

- AttrQuality.ATTR_VALID
- AttrQuality.ATTR_INVALID
- AttrQuality.ATTR_ALARM

EventType enumeration values are mapped to the following integer constants:

- EventType.CHANGE_EVENT
- EventType.QUALITY_EVENT
- EventType.PERIODIC_EVENT
- EventType.ARCHIVE_EVENT
- EventType.USER_EVENT

ArgType enumeration maps Tango data types to python:

- CmdArgType.DEV_VOID
- CmdArgType.DEV_BOOLEAN
- CmdArgType.DEV_SHORT
- CmdArgType.DEV_LONG
- CmdArgType.DEV_FLOAT
- CmdArgType.DEV_DOUBLE
- CmdArgType.DEV_USHORT

- CmdArgType.DEV_ULONG
- CmdArgType.DEV_STRING
- CmdArgType.DEVVAR_CHARARRAY
- CmdArgType.DEVVAR_SHORTARRAY
- CmdArgType.DEVVAR_LONGARRAY
- CmdArgType.DEVVAR_FLOATARRAY
- CmdArgType.DEVVAR_DOUBLEARRAY
- CmdArgType.DEVVAR_USHORTARRAY
- CmdArgType.DEVVAR_ULONGARRAY
- CmdArgType.DEVVAR_STRINGARRAY
- CmdArgType.DEVVAR_LONGSTRINGARRAY
- CmdArgType.DEVVAR_DOUBLESTRINGARRAY
- CmdArgType.DEV_STATE
- CmdArgType.DEVVAR_BOOLEANARRAY
- CmdArgType.DEV_UCHAR

DevSource enumeration values are mapped to the following integer constants:

- DevSource.CACHE
- DevSource.CACHE_DEV
- DevSource.DEV

DevState enumeration values are mapped to the following integer constants:

- DevState.ALARM
- DevState.CLOSE
- DevState.DISABLE
- DevState.EXTRACT
- DevState.FAULT
- DevState.INSERT
- DevState.MOVING
- DevState.OFF
- DevState.ON
- DevState.OPEN
- DevState.RUNNING
- DevState.STANDBY
- DevState.UNKNOWN

cb_sub_model enumeration values are mapped to the following integer constants:

- cb_sub_model.PULL_CALLBACK
- cb_sub_model.PUSH_CALLBACK

AttReqType enumeration values are mapped to the following integer constants:

- AttReqType.READ_REQ
- AttReqType.WRITE_REQ

SerialModel enumeration values are mapped to the following integer constants:

- SerialModel.BY_DEVICE
- SerialModel.BY_CLASS
- SerialModel.BY_PROCESS
- SerialModel.NO_SYNC

3.1 DeviceInfo

DeviceInfo is a Python object containing available information for a device in the following field members

- dev_class: string
- server_id: string
- server_host: string
- server_version: integer
- doc_url: string

3.2 DbDevImportInfo

DbDevImportInfo is a Python object containing information that can be imported from the configuration database for a device in the following field members

- name: device name
- exported: 1 if device is running, 0 otherwise
- ior: CORBA reference of the device
- version: string

3.3 CommandInfo

A device command info with the following members

- cmd_name: command name as ascci string
- cmd_tag: command as binary value (for TACO)
- in_type: input type as binary value (integer)
- out_type: output type as binary value (integer)
- in_type_desc: description of input type (optional)
- out_type_desc: description of output type (optional)
- disp_level: command display level (DispLevel type)

3.4 DevError

Python structure discribing any error resulting from a command execution or an attribute query, with following members

- reason: string
- severity: one of ErrSeverity.WARN, ERRSeverity.ERR or ErrSeverity.PANIC constants
- desc: error description (string)
- out_type: output type as binary value (integer)
- origin: Tango server method in which the error happened

3.5 TimeVal

Time value structure with three field members

- tv_sec: seconds
- tv_usec: microseconds
- ty_nsec: nanoseconds

3.6 DeviceDataHistory

A python object obtained as a result of query of a command history with field members

- time: time of command execution (see TimeVal type)
- cmd_failed: true if attribute command execution failed
- value: returned value as a python object, valid if cmd_failed is false,
- errors: list of errors that occurred (see DevError type) empty if cmd_failed is false

3.7 AttributeInfo

A structure containing available information for an attribute with the following members

- name: attribute name
- writable: one of AttrWriteType constant values AttrWriteType.READ,
  AttrWriteType.READ_WITH_WRITE,
  AttrWriteType.WRITE
  or AttrWriteType.READ_WRITE
- data_format: one of AttrDataFormat constant values AttrWriteType.SCALAR,
  AttrWriteType.SPECTRUM,
  or AttrWriteType.IMAGE
- data_type: integer value indicating attribute type (float, string,..)
- max_dim_x: first dimension of attribute (spectrum or image attributes)
- max_dim_y: second dimension of attribute(image attribute)
- description: string describing the attribute
- label: attribute label (Voltage, time, ...)
- unit: attribute unit (V, ms, ...)
- standard_unit: string
- display_unit: string

- format: string
- min_value: string
- max_value: string
- min_alarm: string
- max_alarm: string
- writable_attr_name: string
- extensions: list of strings
- disp_level: one of DispLevel constants DispLevel.OPERATOR or DispLevel.EXPERT

3.8 AttributeValue

A structure encapsulating the attribute value with additional information in the following members

- value: python object with effectuve value
- quality: one of AttrQuality constant values AttrQuality.VALID, AttrQuality.INVALID or AttrQuality.ALARM
- time: time of value read (see TimeVal type)
- name: attribute name
- dim_x: effective first dimension of attribute (spectrum or image attributes)
- dim_y: effective second dimension of attribute(image attribute)

3.9 DeviceAttributeHistory

A python object obtained as a result of an attribute read history query with field members

- attr_failed: true if attribute read operation failed
- value: attribute value as an AttributeValue type valid if attr_failed is false
- errors: list of errors that occurred (see DevError type) empty if attr_failed is false
- name: attribute name
- dim_x: effective first dimension of attribute (spectrum or image attributes)
- dim_y: effective second dimension of attribute(image attribute)

3.10 EventData

A python object containing data generated by an event; it has the following members

- attr_name: the name of the atribute that generated the event
- attr_value: an AttributeValue instance
- device: a string containing the name of the device that generated the event
- err: True if an error occurred, False otherwise
- errors: a DevErrorList instance
- event: a value of the EventType enum

3.11 DeviceProxy

DeviceProxy is the high level Tango object which provides the client with an easy to use interface to TANGO devices. DeviceProxy provides interfaces to all TANGO Device interfaces. The DeviceProxy manages timouts, stateless connections and reconnection if the device server is restarted. To create a DeviceProxy, a Tango Device name must be set in the object constructor.

```
dev = DeviceProxy("tango/tangotest/1")
```

3.11.1 state()

A method which returns the state of the device.

- Parameters: None
- Return : DevState constant
- Example:

```
dev = DeviceProxy("tango/tangotest/1")
if dev.state() == DevState.ON: ...
```

3.11.2 status()

A method which returns the status of the device as a string.

- Parameters: None
- Return : string

3.11.3 ping()

A method which sends a ping to the device

- Parameters: None
- Return : time elapsed in milliseconds

3.11.4 set_timeout_millis()

Set client side timeout for device in milliseconds. Any method which takes longer than this time to execute will throw an exception.

- Parameters:
  - timeout: integer value of timeout in milliseconds
- Return : None
- Example:

```
dev.set_timeout_millis(1000)
```

3.11.5 get_timeout_millis()

Get the client side timeout in milliseconds.

- Parameters: None

3.11.6 get_idl_version()

Get the version of the Tango Device interface implemented by the device.

3.11.7 set_source()

Set the data source(device, polling buffer, polling buffer then device) for command_inout and read_attribute methods.

- Parameters:
  - source: DevSource constant
- Return : None
- Example:

```
dev.set_source(DevSource.CACHE_DEV)
```

3.11.8 get_source()

Get the data source(device, polling buffer, polling buffer then device) used by command_inout or read_attribute methods.

- Parameters: None
- Return : DevSource constant
- Example:

```
source = dev.get_source()
if source == DevSource.CACHE_DEV: ...
```

3.11.9 black_box()

Get the last commands executed on the device server.

- Parameters:
  - n: n number of commands to get"
- Return : list of strings containing the date, time, command and from which client computer the command was executed.

3.11.10 name()

Return the device name from the device itself.

3.11.11 adm_name()

Return the name of the corresponding administrator device. This is useful if you need to send an administration command to the device server, e.g restart it.

3.11.12 dev_name()

Return the device name as it is stored locally.

3.11.13 alias()

Return the device alias name or throws an exception if no alias is defined

3.11.14 info()

A method which returns information on the device

- Parameters: None
- Return : DeviceInfo object
- Example:

```
dev_info = dev.info()
print dev_info.dev_class
print dev_info.server_id
print dev_info.server_host
print dev_info.server_version
print dev_info.doc_url
print dev_info.dev_type
```

All DeviceInfo fields are strings except for the server_version which is an integer.

3.11.15 import_info()

Query the device for import info from the database.

- Parameters: None
- Return : DbDevImportInfo object
- Example:

```
dev_import = dev.import_info()
print dev_import.name
print dev_import.exported
print dev_ior.ior
print dev_version.version
```

All DbDevImportInfo fields are strings except for exported which is an integer.

3.11.16 description()

Get device description.

- Parameters: None
- Return : string describing the device

3.11.17 command_query()

Query the device for information about a single command.

- Parameters:
  - command: command name
- Return : CommandInfo object
- Example:

```
com_info = dev.command_query("DevString")
print com_info.cmd_name
print com_info.cmd_tag
print com_info.in_type
print com_info.out_type
print com_info.in_type_desc
print com_info.out_type_desc
print com_info.disp_level
```

3.11.18 command_list_query()

Query the device for information on all commands.

- Parameters: None
- Return : list of CommandInfo objects

3.11.19 command_inout()

Execute a command, on a device, which takes zero or one input argument.

- Parameters:
  - name: command name
  - argin: if needed, python object containing the input argument
- Return : result of command execution as a python object
- Example:

```
str_res = dev.command_inout("DevString","Hello!")
print str_res
```

3.11.20 command_history()

Retrive command history from the command polling buffer.

- Parameters:
  - name: command name
  - depth: integer representing the wanted history depth
- Return : a list of DeviceDataHistory objects.
- Example:

```
com_hist = dev.command_history("DevString",3)
for dev_hist in com_hist: print dev_hist
```

See DeviceDataHistory documentation form more detail.

3.11.21 attribute_query()

Query the device for information about a single attribute.

- Parameters:
  - attribute: attribute name
- Return : AttributeInfo object
- Example:

```
attr_info = dev.attribute_query("short_scalar")
print attr_info.name
print attr_info.writable
print attr_info.data_format
print attr_info.data_type
print attr_info.max_dim_x
print attr_info.max_dim_y
print attr_info.description
print attr_info.label
print attr_info.unit
```

```
print attr_info.standard_unit
print attr_info.display_unit
print attr_info.format
print attr_info.min_value
print attr_info.max_value
print attr_info.min_alarm
print attr_info.max_alarm
print attr_info.writable_attr_name
print attr_info.extensions
print attr_info.disp_level
```

See AttributeInfo documentation form more detail.

3.11.22 attribute_list_query()

Query the device for information on a list of attributes.

- Parameters: None
- Return : list of AttributeInfo objects.

3.11.23 get_attribute_list()

Return the names of all attributes implemented for this device.

- Parameters: None
- Return : list of strings

3.11.24 get_attribute_config()

Same effect as calling attribute_query() or attribute_list_query(), depending on the argument passed to the method: a string name or a list of names.

3.11.25 set_attribute_config()

Change the attribute configuration for the specified attributes.

- Parameters: list of AttributeInfo types
- Return : None

3.11.26 read_attribute()

Read a single attribute.

- Parameters:
  - name: attribute name
- Return : AttributeValue object
- Example:

```
attr_val = dev.read_attribute("short_scalar")
print attr_val.value
print attr_val.time
print attr_val.quality
print attr_val.name
print attr_val.dim_x
print attr_val.dim_y
```

See AttributeValue documentation string form more detail.

3.11.27 read_attributes()

Read the list of specified attributes.

- Parameters:
  - name: list of attribute names
- Return : list of AttributeValue types

3.11.28 write_attribute()

Write the specified attribute.

- Parameters:
  - attr_val: AttributeValue type
- Return : None
- Example:

```
attr_val = dev.read_attribute("short_scalar")
attr_val.value = 5
dev.write_attribute(attr_val)
```

3.11.29 write_attributes()

Write the specified list of attributes.

- Parameters:
  - attr_list: list of AttributeValue objects
- Return : None

3.11.30 attribute_history()

Retrive attribute history from the command polling buffer.

- Parameters:
  - name: attribute name
  - depth: integer representing the wanted history depth
- Return : a list of DeviceAttributeHistory types
- Example:

```
for dev_hist in dev.attribute_history("short_scalar",3):
print dev_hist
```

See DeviceAttributeHistory documentation string form more detail.

3.11.31 is_command_polled()

True if the command is polled.

- Parameters:
  - cmd_name: command name
- Return : boolean value

3.11.32 is_attribute_polled()

True if the attribute is polled.

- Parameters:
  - attr_name: command name
- Return : boolean value

3.11.33 get_command_poll_period()

Return the command polling period.

- Parameters:
  - cmd_name: command name
- Return : polling period in milliseconds

3.11.34 get_attribute_poll_period()

Return the attribute polling period.

- Parameters:
  - attr_name: attribute name
- Return : polling period in milliseconds

3.11.35 polling_status()

- Return the device polling status.
- Parameters:None
- Return : list of strings, with one string for each polled command/attribute. Each string is a multi-line string with
  - attribute/command name
  - attribute/command polling period in milliseconds
  - attribute/command polling ring buffer
  - time needed for last attribute/command execution in milliseconds
  - time since data in the ring buffer has not been updated
  - delta time between the last records in the ring buffer
  - exception parameters in case of the last execution failed

3.11.36 poll_command()

Add a command to the list of polled commands.

- Parameters:
  - cmd_name: command name
  - period: polling period in milliseconds
- Return : None

3.11.37 poll_attribute()

Add an attribute to the list of polled attributes.

- Parameters:
  - attr_name: attribute name
  - period: polling period in milliseconds
- Return : None

3.11.38 stop_poll_command()

Remove a command from the list of polled commands.

- Parameters:
  - cmd_name: command name
- Return : None

3.11.39 stop_poll_attribute()

Remove an attribute from the list of polled attributes.

- Parameters:
  - attr_name: attribute name
- Return : None

3.11.40 get_property()

Get a list of properties for a device.

- Parameters:
  - prop_list: list of property names
- Return : a dictionary which keys are the property names the value associated with each key being a list of strings representing the property value

3.11.41 put_property()

Put a list of properties for a device.

- Parameters:
  - props: a dictionary which keys are the property names the value associated with each key being a list of strings representing the property value
- Return : None

3.11.42 delete_property()

Delete a list of properties for a device.

- Parameters:
  - prop_list: list of property names
- Return : None

3.11.43 subscribe_event()

Subscribes to an event generated by the device

- Parameters:
  - attr_name: a string containing the attribute name to subscribe to
  - event: a value of EventType enum
  - call: the callback object that implements the push_event
  - filters: a list of strings containing the filters
- Return : the id of the event
- Example :

```
class PyCallback:
   def push_event(self,event):
      if not event.err:
         print event.attr_name, event.attr_value.value
      else:
         print event.errors

cb = PyCallback();
ev = dev.subscribe_event('long_scalar', EventType.CHANGE, cb, [])
```

3.11.44 unsubscribe_event()

Unsubscribes from a given event

- Parameters: the id of the event
- Return : None

3.12 AttributeProxy

AttributeProxy is the high level Tango object which provides the client with an easy to use interface to TANGO attributes. The AttributeProxy manages timeouts, stateless connections and reconnection if the device server is restarted. To create AttributeProxy instance, two constructors are provided which take as arguments:

- A fully qualified Tango attribute name (device_name/attribute_name)
- An already constructed DeviceProxy instance and the attribute name. This constructor will do a deep copy of the DeviceProxy instance.

See test_att_proxy.py for more examples of AttributeProxy class usage.

```
att = AttributeProxy("tango/tangotest/1/short_scalar")
dev = DeviceProxy("tango/tangotest/1")
another_att = AttributeProxy(dev,"short_scalar")
```

3.12.1 state()

A method which returns the state of the device.

- Parameters: None
- Return : DevState constant
- Example:

```
att = AttributeProxy("tango/tangotest/1/long_scalar")
if att.state() == DevState.ON: ...
```

3.12.2 status()

A method which returns the status of the device as a string.

- Parameters: None
- Return : string

3.12.3 ping()

A method which sends a ping to the device

- Parameters: None
- Return : time elapsed in milliseconds

3.12.4 name()

Return the attribute name.

3.12.5 get_device_proxy()

A method which returns a proxy to the device associated with the attribute.

- Parameters: None
- Return : DeviceProxy object

3.12.6 set_transparency_reconnection()

A method to enable transparency reconnection in case the underlying device has been restarted

- Parameters: A boolean set to True if you want transparency reconnection
- Return: Nothing

3.12.7 get_transparency_reconnection()

A method to get the transparency reconnection flag value

- Parameters: None
- Return: Boolean

3.12.8 get_config()

Query the attribute configuration.

- Parameters: None
- Return : AttributeInfo object
- Example:

```
attr_info = att.get_config()
print attr_info.name
print attr_info.writable
print attr_info.data_format
print attr_info.data_type
print attr_info.max_dim_x
print attr_info.max_dim_y
print attr_info.description
print attr_info.label
print attr_info.unit
print attr_info.standard_unit
```

```
print attr_info.display_unit
print attr_info.format
print attr_info.min_value
print attr_info.max_value
print attr_info.min_alarm
print attr_info.max_alarm
print attr_info.writable_attr_name
print attr_info.extensions
print attr_info.disp_level
```

See AttributeInfo documentation form more detail.

3.12.9 set_config()

Change the attribute configuration for the specified attributes.

- Parameters: AttributeInfo
- Return : None

3.12.10 read()

Reads the attribute.

- Parameters: None
- Return : AttributeValue object
- Example:

```
attr_val = dev.read_attribute()
print attr_val.value
print attr_val.time
print attr_val.quality
print attr_val.name
print attr_val.dim_x
print attr_val.dim_y
```

See AttributeValue documentation string form more detail.

3.12.11 read_asynch()

Performs an asynchronous read

- Parameters: None or Callback
- Return : id of the read or None
- Example :

```
# without callback
id = att.read_asynch()
att.read_reply(id).value

# with callback
class PyCallback:
    def attr_read(self,event):
        if not event.err:
    for el in event.argout:
        print el.value
else:
    print event.errors

cb = PyCallback();
ApiUtil().set_asynch_cb_sub_model(cb_sub_model.PUSH_CALLBACK)
att.read_asynch(cb)
```

3.12.12 read_reply()

Checks if the read has been performed

- Parameters:
  - id: id of the read
  - to : timeout in millis (optional)
- Return : AttributeValue instance

3.12.13 write()

Write the attribute.

- Parameters:
  - attr_val: AttributeValue type
- Return : None
- Example:

```
attr_val = dev.read_attribute("short_scalar")
attr_val.value = 5
dev.write_attribute(attr_val)
```

3.12.14 write_asynch()

Todo

3.12.15 write_reply()

Todo

3.12.16 history()

Retrieve attribute history from the command polling buffer.

- Parameters:
  - depth: integer representing the wanted history depth
- Return : a list of DeviceAttributeHistory types
- Example:

```
for dev_hist in dev.attribute_history("short_scalar",3):
print dev_hist
```

See DeviceAttributeHistory documentation string form more detail.

3.12.17 is_polled()

True if the attribute is polled.

- Parameters: None
- Return : boolean value

3.12.18 get_poll_period()

Returns the attribute polling period.

- Parameters: None
- Return : polling period in milliseconds

3.12.19 poll()

Adds the attribute to the list of polled attributes.

- Parameters: None
- Return : None

3.12.20 stop_poll()

Removes the attribute from the list of polled attributes.

- Parameters: None
- Return : None

3.12.21 get_property()

Get a list of properties for the attribute.

- Parameters:
  - prop_list: list of property names
- Return : a dictionary which keys are the property names the value associated with each key being a list of strings representing the property value

3.12.22 put_property()

Put a list of properties for the attribute.

- Parameters:
  - props: a dictionary which keys are the property names the value associated with each key being a list of strings representing the property value
- Return : None

3.12.23 delete_property()

Delete a list of properties for the attribute.

- Parameters:
  - prop_list: list of property names
- Return : None

3.12.24 subscribe_event()

Subscribes to an event generated by the device associated with the attribute

- Parameters:
  - event: a value of EventType enum
  - call: the callback object that implements the push_event
  - filters: a list of strings containing the filters
- Return : the id of the event
- Example :

```
class PyCallback:
   def push_event(self,event):
      if not event.err:
         print event.attr_name, event.attr_value.value
      else:
         print event.errors

cb = PyCallback();
ev = att.subscribe_event(EventType.CHANGE, cb, [])
```

3.12.25 unsubscribe_event()

Unsubscribes from a given event

- Parameters: the id of the event
- Return : None

3.13 ApiUtil

In the current version of the bindings this object wraps some methods of the ApiUtil C++ singleton; if other methods are needed they will be included in the future.

3.13.1 get_asynch_replies()

Awakes waiting callbacks in the PULL_CALLBACK model

- Parameters: None

3.13.2 set_asynch_cb_sub_model()

Sets asynchronous callbcack model

- Parameters: callback model
- Return : None
- Example : see read_asynch

4 The Tango Database Python API

The PyTango allows access from Python environment to the following Tango high level C++ Database classes and structures:

- DbDevInfo
- DbDevImportInfo
- DbDevExportInfo
- Database

4.1 DbDevInfo

A structure containing available information for a device with the following members:

- name: string
- class: string
- server: string

4.2 DbDevExportInfo

Export info for a device with the following members:

- name: device name
- ior: CORBA reference of the device
- host: name of the computer hosting the server
- version: string
- pid: process identifier

4.3 Database

Database is the high level Tango object which contains the link to the static database. Database provides methods for all database commands: get_device_property(), info(), etc. To create a Database, use the default constructor:

```
db = Database()
```

The constructor uses the TANGO_HOST environment variable to determine which instance of the Database to connect to. Alternatively you can specify host and port:

```
db = Database('host', port)
```

4.3.1 get_info()

Query the database for some general info about the tables.

- Parameters: None
- Return : a multi-line string

4.3.2 add_device()

Add a device to the database. The device name, server and class are specified in the DbDevInfo structure.

- Parameters: DbDevInfo structure
- Return : None
- Example:

```
dev_info = DbDevInfo()
dev_info.name = "my/own/device"
dev_info.class = "MyDevice"
dev_info.server = "MyServer/test"
db.add_device(dev_info)
```

4.3.3 delete_device()

Delete the device of the specified name from the database

- Parameters: Device name
- Return : None
- Example:

```
db.delete_device("my/own/device")
```

4.3.4 import_device()

Query the databse for the import info of the specified device.

- Parameters: Device name
- Return : DbDevImportInfo object
- Example:

```
dev_imp_info = db.import_device("my/own/device")
print dev_imp_info.name
print dev_imp_info.exported
print dev_imp_info.ior
print dev_imp_info.version
```

4.3.5 export_device()

Update the export info for this device in the database.

- Parameters: DbDevExportInfo structure
- Return : None
- Example:

```
dev_export = DbDevExportInfo()
dev_export.name = "my/own/device"
dev_export.ior = "the real ior"
dev_export.host = "the host"
dev_export.version = "1.0"
dev_export.pid = "...."
db.export_device(dev_export)
```

4.3.6 unexport_device()

Mark the specified device as unexported in the database.

- Parameters: Device name
- Return : None
- Example:

```
db.unexport_device("my/own/device")
```

4.3.7 add_server()

Add a group of devices to the database.

- Parameters:
  - Server name
  - List of DbDevInfo structures
- Return : None

4.3.8 delete_server()

Delete the device server and its associated devices from database.

- Parameters: Server name
- Return : None

4.3.9 export_server()

Export a group of devices to the database.

- Parameters:
  - Server name
  - List of DbDevExportInfo structures
- Return : None

4.3.10 unexport_server()

Mark all devices exported for this server as unexported.

- Parameters: Server name
- Return : None

4.3.11 get_device_name()

Query the database for a list of devices served by a server for a given device class.

- Parameters:
  - Server name
  - Device class name
- Return : List of device names

4.3.12 get_device_alias()

Query the database for a list of aliases for the specified device.

- Parameters: Device name
- Return : List of aliases

4.3.13 get_device_exported()

Query the database for a list of exported devices whose names satisfy the supplied filter (* is wildcard for any character(s)).

- Parameters: string filter
- Return : List of exported devices

4.3.14 get_device_domain()

Query the database for a list of of device domain names which match the wildcard provided (* is wildcard for any character(s)). Domain names are case insensitive.

- Parameters: string filter
- Return : List of device domain names

4.3.15 get_device_family()

Query the database for a list of of device family names which match the wildcard provided (* is wildcard for any character(s)). Family names are case insensitive.

- Parameters: string filter
- Return : List of device family names

4.3.16 get_device_member()

Query the database for a list of of device member names which match the wildcard provided (* is wildcard for any character(s)). member names are case insensitive.

- Parameters: string filter
- Return : List of device member names

4.3.17 get_property()

Query the database for a list of object (i.e non-device) properties.

- Parameters:
  - string: object name
  - prop_list: list of property names
- Return : a dictionary which keys are the property names the value associated with each key being a list of strings representing the property value

4.3.18 put_property()

Insert or update a list of properties for the specified object.

- Parameters:
  - string: object name
  - props: a dictionary which keys are the property names the value associated with each key being a list of strings representing the property value
- Return : None

4.3.19 delete_property()

Delete a list of properties for the specified object.

- Parameters:
  - string: object name
  - prop_list: list of property names
- Return : None

4.3.20 get_device_property()

Query the database for a list of device properties.

- Parameters:
  - string: device name
  - prop_list: list of property names
- Return : a dictionary which keys are the property names the value associated with each key being a list of strings representing the property value

4.3.21 put_device_property()

Insert or update a list of properties for the specified device.

- Parameters:
  - string: device name
  - props: a dictionary which keys are the property names the value associated with each key being a list of strings representing the property value
- Return : None
- Example :

```
properties_logging={'logging\_level':["DEBUG"],'logging\_target':["device::tmp/log/1"]}
db.put_device_property("px1/tdl/mouse",properties_logging)
```

4.3.22 delete_device_property()

Delete a list of properties for the specified device.

- Parameters:
  - string: device name
  - prop_list: list of property names
- Return : None

4.3.23 get_device_attribute_property()

Query the database for a list of device attribute properties.

- Parameters:
  - string: device name
  - prop_list: list of attribute names
- Return : a dictionary which keys are the attribute names, the value associated with each key being another dictionary which keys are the attribute property names.

4.3.24 put_device_attribute_property()

Insert or update a list of properties for the specified device attribute.

- Parameters:
  - string: device name
  - props: a dictionary which keys are the attribute names, the value associated with each key being another dictionary which keys are the attribute property names.
- Return : None
- Example :

```
properties_positionX={"min_value": ["20"],
                      "max_value": ["1000"],
                      "min_alarm":["50"],
                      "max_alarm":["950"],
                      "label":["Pos en pixel"],
                      "format":["%3d"]}
properties_positionY={"min_value": ["20"],
                      "max_value": ["1000"],
                      "min_alarm":["50"],
```

```
                        "max_alarm":["950"],
                        "label":["Pos en pixel"],
                        "format":["%3d"]}
    attr_props_mouse = { "positionX": properties_positionX ,
                        "positionY": properties_positionY }
    db.put_device_attribute_property(mouse_name, attr_props_mouse)
```

4.3.25 delete_device_attribute_property()

Delete a list of properties for the specified device attribute.

- Parameters:
  - string: device name
  - prop_list: list of attribute names
- Return : None

4.3.26 get_class_property()

Query the database for a list of device class properties.

- Parameters:
  - string: class name
  - prop_list: list of property names
- Return : a dictionary which keys are the property names the value associated with each key being a list of strings representing the property value

4.3.27 put_class_property()

Insert or update a list of properties for the specified device class.

- Parameters:
  - string: class name
  - props: a dictionary which keys are the property names the value associated with each key being a list of strings representing the property value
- Return : None

4.3.28 delete_class_property()

Delete a list of properties for the specified device class.

- Parameters:
  - string: class name
  - prop_list: list of property names
- Return : None

4.3.29 get_class_attribute_property()

Query the database for a list of device class attribute properties.

- Parameters:
  - string: class name
  - prop_list: list of property names
- Return : a dictionary which keys are the attribute names, the value associated with each key being another dictionary which keys are the attribute property names.

4.3.30 put_class_attribute_property()

Insert or update a list of properties for the specified class attribute.

- Parameters:
  - string: class attribute name
  - props: a dictionary which keys are the attribute names, the value associated with each key being another dictionary which keys are the attribute property names.
- Return : None

4.3.31 delete_class_attribute_property()

Delete a list of properties for the specified class attribute.

- Parameters:
  - string: class name
  - prop_list: list of attribute names
- Return : None

5 Tango Device Server in Python

This chapter does not explain what a Tango device or a device server is. This is explained in details in "The Tango control system manual" available at http://www.esrf.fr/computing/cs/tango/tango_doc/kernel_doc/ds_prog/tango.

The device server we will detailed in the following example is a Tango device server with one Tango class called PyDsExp. This class has two commands called IOLong and IOStringArray and two attributes called Long_attr and Short_attr_rw.

5.1 Importing python modules

To write a Python script which is a Tango device server, you need to import two modules which are :

1. The **PyTango** module which is the Python to C++ interface
2. The Python classical **sys** module

This could be done with code like (supposing the PYTHONPATH environment variable is correctly set)

```
1
2 import PyTango
3 import sys
```

5.2 The main part of a Python device server

The rule of this part of a Tango device server is to:

- Create the PyUtil object passing it the Python interpreter command line arguments
- Add to this object the list of Tango class(es) which have to be hosted by this interpreter
- Initialise the device server
- Run the device server loop

The following is a typical code for this main function

```
1
2  if __name__ == '__main__':
3     py = PyTango.PyUtil(sys.argv)
4     py.add_TgClass(PyDsExpClass,PyDsExp,'PyDsExp')
5
6     U = PyTango.Util.instance()
7     U.server_init()
8     U.server_run()
```

Line 3 : Create the PyUtil object passing it the interpreter command line arguments
Line 4 : Add the Tango class 'PyDsExp' to the device server. The *add_ TgClass()* method of the PyUtil class has three arguments which are the Tango class PyDsExpClass instance, the Tango PyDsExp instance and the Tango class name
Line 7 : Initialise the Tango device server
Line 8 : Run the device server loop

5.3 The PyDsExpClass class in Python

The rule of this class is to :

- Host and manage data you have only once for the Tango class whatever devices of this class will be created
- Define Tango class command(s)
- Define Tango class attribute(s)

In our example, the code of this Python class looks like:

```
1  class PyDsExpClass(PyTango.PyDeviceClass):
2
3      cmd_list = {'IOLong':[[PyTango.DevLong,"Number"],
4                             [PyTango.DevLong,"Number * 2"]],
5                  'IOStringArray':[[PyTango.DevVarStringArray,"Array of string"],
6                        [PyTango.DevVarStringArray,"This reversed array"]]
7                 }
8
9
10     attr_list = {'Long_attr':[[PyTango.DevLong,
11                                PyTango.SCALAR,
12                                PyTango.READ],
13                               {'min alarm':1000,'max alarm':1500}],
14                  'Short_attr_rw':[[PyTango.DevShort,
15                                    PyTango.SCALAR,
16                                    PyTango.READ_WRITE]]
17                }
```

Line 1 : The PyDsExpClass class has to inherit from the PyTango.PyDeviceClass class
Line 3 to 7 : Definition of the **cmd_list** dictionnary defining commands. The IOLong command is defined at lines 3 and 4. The IOStringArray command is defined in line 5 and 6
Line 10 to 16 : Definition of the **attr_list** dictionnary defining attributes. The Long_attr attribute is defined at lines 10 to 13 and the Short_attr_rw attribute is defined at lines 14 to 16

If you have something specific to do in the class constructor like initializing some specific data member, you will have to code a class constructor. An example of such a contructor is :

```
1 class PyDsExpClass(PyTango.PyDeviceClass):
2     def __init__(self,name):
3        PyTango.PyDeviceClass.__init__(self,name)
4        self.set_type("TestDevice")
```

The device type is set at line 4.

5.3.1 Defining commands

As shown in the previous example, commands have to be defined in a dictionary called **cmd_list** as a data member of the xxxClass class of the Tango class. This dictionary has one element per command. The element key is the command name. The element value is a Python list which defines the command. The generic form of a command definition is

'cmd_name':[[in_type,<"In desc">],[out_type,<"Out desc">],<{opt parameters}>]

The first element of the value list is itself a list with the command input data type (one of the PyTango.ArgType pseudo enumeration value) and optionally a string describing this input argument. The second element of the value list is also a list with the command output data type (one of the PyTango.ArgType pseudo enumeration value) and optionally a string describing it. These two elements are mandatory. The third list element is optional and allows additional command definition. The authorized element for this dictionary are summarized in the following array

key	value	definition
"display type"	PyTango.DispLevel enum value	The command display type
"polling period"	Any number	The command polling period (mS)
"default command"	True or False	To define that it is the default command

5.3.2 Defining attributes

As shown in the previous example, attributes have to be defined in a dictionary called **attr_list** as a data member of the xxxClass class of the Tango class. This dictionary has one element per attribute. The element key is the attribute name. The element value is a Python list which defines the attribute. The generic form of an attribute definition is

'attr_name':[[mandatory parameters],<{opt parameters}>]

For any kind of attributes, the mandatory parameters are:

[attr data type, attr data format, attr data R/W type]

The attribute data type is one of the possible value for attributes of the PyTango.ArgType pseudo enunmeration. The attribute data format is one of the possible value of the PyTango.AttrDataFormat pseudo enumeration and the attribute R/W type is one of the possible value of the PyTango.AttrWriteType pseudo enumeration. For spectrum attribute, you have to add the maximum X size (a number). For image attribute, you have to add the maximun X and Y dimension (two numbers). The authorized elements for the dictionnary defining optional parameters are summarized in the following array

key	value	definition
"display type"	PyTango.DispLevel enum value	The attribute display type
"polling period"	Any number	The attribute polling period (mS)
"memorized"	True or True_without_hard_applied	Define if and how the att. is memorized
"label"	A string	The attribute label
"description"	A string	The attribute description
"unit"	A string	The attribute unit
"standard unit"	A number	The attribute standard unit
"display unit"	A string	The attribute display unit
"format"	A string	The attribute display format
"max value"	A number	The attribute max value
"min value"	A number	The attribute min value
"max alarm"	A number	The attribute max alarm
"min alarm"	A number	The attribute min alarm
"delta time"	A number	The attribute RDS alarm delta time
"delta val"	A number	The attribute RDS alarm delta val

5.4 The PyDsExp class in Python

The rule of this class is to implement methods executed by commands and attributes. In our example, the code of this class looks like :

```
1  class PyDsExp(PyTango.Device_3Impl):
2      def __init__(self,cl,name):
3          PyTango.Device_3Impl.__init__(self,cl,name)
4          print 'In PyDsExp __init__'
5          PyDsExp.init_device(self)
6
7      def init_device(self):
8          print 'In Python init_device method'
9          self.set_state(PyTango.DevState.ON)
10         self.attr_short_rw = 66
11         self.attr_long = 1246
12
13 #------------------------------------------------------------------
14
15     def delete_device(self):
16         print "[Device delete_device method] for device",self.get_name()
17
18 #------------------------------------------------------------------
19
20     def is_IOLong_allowed(self):
21         if (self.get_state() == PyTango.DevState.ON):
22             return True
23         else:
24             return False
25
26     def IOLong(self,in_data):
27         print "[IOLong::execute] received number",in_data
28         return in_data * 2;
29
30 #------------------------------------------------------------------
31
32     def is_IOStringArray_allowed(self):
33         if (self.get_state() == PyTango.DevState.ON):
34             return True
35         else:
36             return False
37
38     def IOStringArray(self,in_data):
39         out_data=in_data
40         out_data.reverse()
41         self.y = out_data
42         return out_data
43
44 #------------------------------------------------------------------
45
46     def read_attr_hardware(self,data):
47         print 'In read_attr_hardware'
48
49 #------------------------------------------------------------------
50
51     def read_Long_attr(self,the_att):
52         print "[PyDsExp::read_attr] attribute name Long_attr"
53         the_att.set_value(self.attr_long)
54
55 #------------------------------------------------------------------
56
57     def read_Short_attr_rw(self,the_att):
58         print "[PyDsExp::read_attr] attribute name Short_attr_rw"
59         the_att.set_value(self.attr_short_rw)
```

```
60
61  #------------------------------------------------------------------
62
63      def write_Short_attr_rw(self,the_att):
64          print "In write_Short_attr_rw for attribute ",the_att.get_name()
65          data=[]
66          the_att.get_write_value(data)
67          self.attr_short_rw = data[0]
```

Line 1 : The PyDsExp class has to inherit from the PyTango.Device_3Impl
Line 2 to 5 : PyDsExp class constructor. Note that at line 5, it calls the *init_device()* method
Line 7 to 11 : The *init_device()* method. It sets the device state (line 9) and initialises some data members
Line 15 to 16 : The *delete_device()* method. This method is not mandatory. You define it only if you have to do something specific before the device is destroyed
Line 20 to 28 : The two methods for the IOLong command. The first method is called is_IOLong_allowed() and it is the command is_allowed method (line 20 to 24). The second method has the same name than the command name. It is the method which executes the command. The command input data type is a Tango long and therefore, this method receives a Python integer.
Line 32 to 42 : The two methods for the IOStringArray command. The first method is its is_allowed method (Line 32 to 36). The second one is the command execution method (Line 38 to 42). The command input data type is a String array. Therefore, the method receives the array in a Python list of Python strings.
Line 46 to 47 : The *read_attr_hardware()* method. Its argument is a Python list of Python integer.
Line 51 to 53 : The method executed when the Long_attr attribute is read. Note that it sets the attribute value with the *PyTango.set_attribute_value()* function.
Line 57 to 59 : The method executed when the Short_attr_rw attribute is read.
Line 63 to 67 : The method executed when the Short_attr_rw attribute is written. It gets the attribute value with a call to the Attribute method *get_write_value()* with a list as argument.

5.4.1 General methods

The following array summarizes how the general methods we have in a Tango device server are implemented in Python.

Name	Input par (with "self")	return value	mandatory
init_device	None	None	Yes
delete_device	None	None	No
always_executed_hook	None	None	No
signal_handler	Python integer	None	No
read_attr_hardware	Python list of integer	None	No

5.4.2 Implementing a command

Commands are defined as described in chapter 5.3.1. Nevertheless, some methods implementing them have to be written. These methods names are fixed and depend on command name. They have to be called

is_<Cmd_name>_allowed and **<Cmd_name>**

For instance, with a command called MyCmd, its is_allowed method has to be called is_MyCmd_allowed and its execution method has to be called simply MyCmd. The following array gives some more info on these methods.

Name	Input par (with "self")	return value	mandatory
is_<Cmd_name>_allowed	None	Python boolean	No
Cmd_name	Depends on cmd type	Depends on cmd type	Yes

Tango has more data types than Python which is more dynamic. How Tango data are transferred to Python method implementing commands is described in the following array:

Tango data type	Python type
DEV_VOID	No data
DEV_BOOLEAN	A Python boolean
DEV_SHORT	A Python integer
DEV_LONG	A Python integer
DEV_FLOAT	A Python float
DEV_DOUBLE	A Python float
DEV_USHORT	A Python integer
DEV_ULONG	A Python integer
DEV_STRING	A python string - See Note
DEVVAR_CHARARRAY	A Python list of Python integer - See Note
DEVVAR_SHORTARRAY	A Python list of Python integer - See Note
DEVVAR_LONGARRAY	A Python list of Python integer - See Note
DEVVAR_FLOATARRAY	A Python list of Python float - See Note
DEVVAR_DOUBLEARRAY	A Python list of Python float - See Note
DEVVAR_USHORTARRAY	A Python list of Python integer - See Note
DEVVAR_ULONGARRAY	A Python list of Python integer - See Note
DEVVAR_STRINGARRAY	A Python list of Python string - See Note
DEVVAR_LONGSTRINGARRAY	A Python tuple with two elements (See Note) : 1 - A Python list of Python integer 2 - A Python list of Python string
DEVVAR_DOUBLESTRINGARRAY	A Python tuple with two elements (See Note) : 1 - A Python list of Python float 2 - A Python list of Python string

Note : For all commands returning data of this type, Boost.Python need a Python object with a reference count of at least 2 otherwise, you will have an error like "ReferenceError: Attempt to return dangling pointer to object of type: xxx". The following code is an example of a command returning a Tango::DEVVAR_SHORTARRAY data type (A list in Python)

```
1 def IOShortArray(self,in_data):
2     l = range(len(in_data))
3     for i in l:
4         in_data[i] = in_data[i] * 2
5     self.dummy = in_data
6     return in_data
```

The python reference count of the returned list is incremented at line 5

5.4.3 Implementing an attribute

Attributes are defined as described in chapter 5.3.2. Nevertheless, some methods implementing them have to be written. These methods names are fixed and depend on attribute name. They have to be called

is_<Attr_name>_allowed and **read_<Attr_name>** or/and **write_<Attr_name>**

For instance, with an attribute called MyAttr, its is_allowed method has to be called is_MyAttr_allowed, its read method has to be called read_MyAttr and its write method has to be called write_MyAttr. The following array gives some more info on these methods.

Name	Input par (with self)	return value	mandatory
is_<Attr_name>_allowed	Request type (AttReqType)	Python boolean	No
read_<Attr_name>	The attribute instance	None	Depend on Attr. type
write_<Attr_name>	The attribute instance	None	Depend on Attr. type

Tango has more data types than Python which is more dynamic. How Tango data are transferred to Python method implementing attributes is described in the following array:

Attr. data format	Attr. data type	Python type
	DEV_BOOLEAN	Python boolean
	DEV_UCHAR	Python integer
	DEV_SHORT	Python integer
SCALAR	DEV_USHORT	Python integer
	DEV_LONG	Python integer
	DEV_FLOAT	Python float
	DEV_DOUBLE	Python float
	DEV_STRING	Python string
	DEV_BOOLEAN	Python list of Python boolean
	DEV_UCHAR	Python list of Python integer
SPECTRUM	DEV_SHORT	Python list of Python integer
or	DEV_USHORT	Python list of Python integer
IMAGE	DEV_LONG	Python list of Python integer
	DEV_FLOAT	Python list of Python float
	DEV_DOUBLE	Python list of Python float
	DEV_STRING	Python list of Python string

The following code is an example of how you write code executed when a client read an attribute which is called Long_attr

```
1   def read_Long_attr(self,the_att):
2      print "[PyDsExp::read_attr] attribute name Long_attr"
3      the_att.set_value(self.attr_long)
```

Line 1 : Method declaration with "the_att" being an instance of the Attribute class representing the Long_attr attribute
Line 3 : Set the attribute value using the method *set_attribute_value()* with the attribute value as parameter

The following code is an example of how you write code executed when a client write the Short_attr_rw attribute

```
1   def write_Short_attr_rw(self,the_att):
2      print "In write_Short_attr_rw for attribute ",the_att.get_name()
3      data=[]
4      the_att.get_write_value(data)
5      self.attr_short_rw = data[0]
```

Line 1 : Method declaration with "the_att" being an instance of the Attribute class representing the Short_attr_rw attribute
Line 3 : Define an empty Python list. This is needed even for scalar attribute to get the value sent by the client
Line 4 : Get the value sent by the client using the method *get_write_value()* with the list defined previously as parameter. This list will be initialised with the value sent by the client.
Line 5 : Store the value written in the device object. Our attribute is a scalar attribute and its value is in the first element of the list.

5.5 Tango C++ wrapped class usable in a Python device server

This chapter is not a precise definition of the method / functions parameters. In the PyTango module source distribution, there is an example of Python device server (DevTest.py) in its "test" directory where all these methods are used. They have exactly the same rules than the C++ one.

5.5.1 The Util class

This class has the following wrapped methods available in Python script

- *instance()* which is a static method
- *server_init()*
- *server_run()*
- *set_serial_model()*

- *get_device_by_name()*
- *get_dserver_device()*
- *get_device_list_by_class()*
- *trigger_cmd_polling()*
- *trigger_attr-polling()*

This class has the following static data members wrapped available in Python script

- _UseDb
- _FileDb

5.5.2 The PyDeviceClass

This class has the following wrapped methods available in Python script

- *get_name()*
- *set_type()*
- *register_signal()*
- *unregister_signal()*
- *signal_handler()*
- *get_cvs_tag()*
- *get_cvs_location()*
- *add_wiz_dev_prop()*
- *add_wiz_class_prop()*

5.5.3 The Device_3Impl class

This class has the following wrapped methods available in Python script

- *set_state()*
- *get_state()*
- *dev_state()*
- *set_status()*
- *get_status()*
- *dev_status()*
- *get_name()*
- *get_device_attr()*
- *get_device_class()*
- *register_signal()*
- *unregister_signal()*
- *signal_handler()*
- *set_change_event()*
- *set_archive_event()*
- *push_change_event()*
- *push_archive_event()*
- *push_event()*
- *add_attribute()*

5.5.4 The MultiAttribute class

This class has the following wrapped methods available in Python script

- *get_w_attr_by_name()*
- *get_w_attr_by_ind()*
- *get_attr_by_name()*
- *get_attr_by_ind()*

5.5.5 The Attribute class

This class has the following wrapped methods available in Python script

- *get_name()*
- *set_quality()*
- *check_alarm()*
- *set_value()*
- *set_value_date_quality()*

5.5.6 The WAttribute class

This class inherits from the Attribute class. It has the following wrapped methods available in Python script

- *get_write_value_length()*
- *get_write_value()*
- *set_write_value()*

5.5.7 The UserDefaultAttrProp class

This class has the following wrapped methods available in Python script

- *set_label()*
- *set_description()*
- *set_format()*
- *set_unit()*
- *set_standard_unit()*
- *set_display_unit()*
- *set_min_value()*
- *set_max_value()*
- *set_min_alarm()*
- *set_max_alarm()*
- *set_min_warning()*
- *set_max_warning()*
- *set_delta_t()*
- *set_delta_val()*
- *set_abs_change()*

- *set_rel_change()*
- *set_period()*
- *set_archive_abs_change()*
- *set_archive_rel_change()*
- *set_archive_period()*

5.5.8 The Attr class

This class has the following wrapped methods available in Python script

- *set_default_properties()*

5.5.9 The SpectrumAttr class

This class inherits from the Attr class

5.5.10 The Image Attr class

This class inherits from the SpectrumAttr class

5.5.11 The vector<DeviceImpl *> class

This class has the following wrapped methods available in Python script

- *at()*
- *size()*

5.5.12 The DServer class

This class inherits from the Device_3Impl class

5.6 Python classes available in the PyTango module

5.6.1 The PyUtil class

This class inherit from the Util class. It hass the following methods

- *PyUtil()* : Its constructor with the interpreter command line arguments as parameter
- *add_TgClass()* : Add a Python Tango class to the device server. This method has three parameters which are :
  - The Tango class xxxClass instance
  - The Tango class xxx instance
  - The Tango class name
- *add_Cpp_TgClass()* : Add a C++ Tango class to the device server. This method has two parameters which are :
  - The Tango class xxxClass name
  - The Tango class name

5.7 Mixing Tango classes (Python and C++) in a Python Tango device server

Within the same python interpreter, it is possible to mix several Tango classes. Here is an example of the main function of a device server with two Tango classes called IRMiror and PLC

```
1
2  if __name__ == '__main__':
3      py = PyTango.PyUtil(sys.argv)
4      py.add_TgClass(PLCClass,PLC,'PLC')
5      py.add_TgClass(IRMirrorClass,IRMirror,'IRMirror')
6
7      U = PyTango.Util.instance()
8      U.server_init()
9      U.server_run()
```

Line 4 : The Tango class PLC is registered in the device server
Line 5 : The Tango class IRMirror is registered in the device server

It is also possible to add C++ Tango class in a Python device server as soon as:

1. The Tango class is in a shared library
2. It exist a C function to create the Tango class.

For a Tango class called **MyTgClass**, the shared library has to be called **MyTgClass.so** and has to be in a directory listed in the LD_LIBRARY_PATH environment variable. The C function creating the Tango class has to be called ***_create_MyTgClass_class()*** and has to take one parameter of type "char *" which is the Tango class name. Here is an example of the main function of the same device server than before but with one C++ Tango class called SerialLine

```
1
2  if __name__ == '__main__':
3     py = PyTango.PyUtil(sys.argv)
4
5     py.add_Cpp_TgClass('SerialLine','SerialLine')
6
7     py.add_TgClass(PLCClass,PLC,'PLC')
8     py.add_TgClass(IRMirrorClass,IRMirror,'IRMirror')
9
10    U = PyTango.Util.instance()
11    U.server_init()
12    U.server_run()
```

Line 5 : The C++ class is registered in the device server
Line 7 and 8 : The two Python classes are registered in the device server

5.8 Debugging a Python Tango device server using Eclipse/PyDev

Python debugger's are "pertubated" if the process you are debugging creates thread(s) without using the Python threading module. This is the case for Tango Python device server where threads are created by the Tango C++ API. If you are using the Eclipse PyDev plug-in, look at their Frequently Asked Questions (FAQ) WEB pages at http://pydev.sourceforge.net/faq.html and search for the question about CORBA program. Follow their two first advices (number 1 and 2) to keep a trace of the PyDev debugging hook and modify your main function like the following :

```
1 if __name__ == '__main__':
2     try:
3         PyTango.PyDev_debug(pydev_hook)
4     except:
5         print "Debugging is not available"
6
7     py = PyTango.PyUtil(sys.argv)
8     py.add_TgClass(PLCClass,PLC,'PLC')
```

```
 9     py.add_TgClass(IRMirrorClass,IRMirror,'IRMirror')
10
11     U = PyTango.Util.instance()
12     U.server_init()
13     U.server_run()
```

Line 1 to 4 : Try to get the global variable "pydev_hook" defined in debugger and call function PyDev_debug of the PyTango with it. If this global is not defined because the script is not running under debugger control, print a message

6 Exception Handling

6.1 Exception definition

All the exceptions that can be thrown by the underlying Tango C++ API are available in the PyTango python module. Hence a user can catch one of the following exceptions: DevFailed, ConnectionFailed, CommunicationFailed, WrongNameSyntax, NonDbDevice, WrongData, NonSupportedFeature, EventSystemFailed. For a detailed meaning and description of the context in which they are thrown, please refer to the Tango control system documentation. When an exception is caught, the sys.exc_info() function returns a tuple of three values that give information about the exception that is currently being handled. The values returned are (type, value, traceback). Since most functions don't need access to the traceback, the best solution is to use something like exctype, value = sys.exc_info()[:2] to extract only the exception type and value. If one of the Tango exceptions is caught, the exctype will be class name of the exception (DevFailed, .. etc) and the value a tuple of dictionary objects all of which containing the following kind of key-value pairs:

- reason: a string describing the error type (more readable than the associated error code)
- desc: a string describing in plain text the reason of the error.
- origin: a string giving the name of the (C++ API) method which thrown the exception
- severity: one of the strings WARN, ERR, PANIC giving severity level of the error.

Example 6.1
```
# Protect the script from Exceptions raised by the Tango or python itself
try:
        # Get proxy on the tangotest1 device
        print "Getting DeviceProxy "
        tangotest = DeviceProxy("tango/tangotest/1")
#Catch Tango and Systems  Exceptions
except DevFailed:
        exctype , value = sys.exc\_info()[:2]
        print "Failed with exception ! " , exctype
        for err in value:
                print " reason" , err["reason"]
                print " description" , err["desc"]
                print "origin" , err["origin"]
                print "severity" , err["severity"]
```

6.2 Throwing exception in a device server

The C++ Tango::Except class with its most important methods have been wrapped to Python. Threfore, in a Python device server, you have the following methods to throw, re-throw or print a Tango::DevFailed exception :

- *throw_exception()* which is a static method
- *re_throw_exception()* which is also a static method
- *print_exception()* which is also a static method

The following code is an example of a command method requesting a command on a sub-device and re-throwing the exception in case of:

```
1   try:
2      dev.command_inout("SubDevCommand")
3   except PyTango.DevFailed,e:
4      PyTango.Except.re_throw_exception(e,"MyClass_CommandFailed",
5                                        "Sub device command SubdevCommand failed",
6                                        "Command()")
```

Line 2 : Send the command to the sub device in a try/catch block
Line 4 - 6 : Re-throw the exception and add a new level of information in the exception stack

7 TODO

- asynch writes in AttributeProxy
- Group object implementation
- Device logging in Python device server
- bugfix
- ...