BAC0 Documentation

Christian Tremblay, P.Eng.

Sep 09, 2024

CONTENTS

CHAPTER

ONE

BAC0

BAC0 is an asynchronous Python 3 (3.10 and over) scripting application that uses BACpypes3 to process BACnet™ messages on an IP network. This library brings out simple commands to browse a BACnet network, read properties from BACnet devices, or write to them.

Python is a simple language to learn and a very powerful tool for data processing. Coupled with BACnet, it becomes a great tool to test devices and interact with controllers.

BAC0 takes its name from the default IP port used by BACnet/IP communication which is port 47808. In hexadecimal, it's written 0xBAC0.

1.1 Test driven development (TDD) for DDC controls

BAC0 is made for building automation system (BAS) programmers. Controllers used in this field are commonly called DDC Controllers (Direct Digital Control).

Typical controllers can be programmed in different ways, depending on the manufacturer selling them (block programming, basic "kinda" scripts, C code, etc…). BAC0 is a unified way, using Python language and BACnet/IP communication, to interact with those controllers once their sequence is built.

BAC0 allows users to simply test an application even if sensors are not connected to the controller. Using the out_of_service property, it's easy to write a value to the input so the controller will think an input is connected.

CHAPTER

TWO

TABLE OF CONTENTS

2.1 Getting started

2.1.1 I know nothing about Python

First, welcome to the Python community. If you're new to Python programming, it can be hard to know where to start.

I highly recommend to start with a complete distribution. That will help you a lot as the majority of important modules will be installed for you.

If you are using Windows, it will simplify your life as some modules needs a C compiler and it can be hard sometimes to compile a module by yourself.

Some examples of complete distributions are Anaconda or Enthought Canopy. As I use Anaconda, I'll focus on this one but you're free to choose the one you prefer.

If you are using a RaspberryPi, have a look to miniconda or berryconda. Both can allow a complete installation of modules like bokeh and Flask. For berryconda, once it's done, run *conda install pandas*. This will install pandas on your RaspberryPi without the need to compile it.

2.1.1.1 Installing a complete distribution

Begin by downloading Anaconda. Install it. Once it's done, you'll get access to a variety of tools like :

- Spyder (and IDE to write the code)
- Anaconda Prompt (a console configured for Python)
- Jupyter Notebook (Python in your browser)
- pip (a script allowing you to install modules)
- conda (a package manager used by Anaconda)

2.1.1.2 Start using pip

Open the Anaconda Prompt (a console terminal with Python configured in the path)

```
pip install BAC0
```

This simple line will look in Pypi (The Python Package Index), download and install everything you need to start using BAC0

2.1.1.3 Check that BAC0 works

In the terminal again, type

```
python -m asyncio
```

This will open a python terminal. In the terminal type

```
>>import BAC0
>>bacnet = BAC0.start()
```

This will show you the installed version. You're good to go.

2.1.2 Where to download the source code

https://github.com/ChristianTremblay/BAC0/

There you'll be able to open issues if you find bugs.

2.1.3 Dependencies

- BAC0 is based on BACpypes3 for all BACnet/IP communication.

  Starting at version 0.9.900, BAC0 will not strictly depend on bokeh or Flask or Pandas to work. Having them will allow to use the complete set of features (the web app with live trending features) but if you don't have them installed, you will be able to use the 'lite' version of BAC0 which is sufficient to interact with BACnet devices.
- It uses Bokeh for Live trending features
- It uses Pandas for every Series and DataFrame (histories)
- It uses Flask to serve the Web app (you will need to pip install flask_bootstrap)

Normally, if you have installed Anaconda, Flask, Bokeh and Pandas will already be installed. You'll only need to install BACpypes3

```
pip install BACpypes3
pip install bokeh (or conda install bokeh if using Anaconda)
```

You're ready to begin using BAC0 !

2.2 How to start BAC0

2.2.1 Intro

BAC0 is a library that will allow you to interact with BACnet devices. It relies on bacpypes3 and take advantages of the features of asyncio to provide fast and efficient communication with BACnet devices.

To start using BAC0, you will need to import the library and create a BAC0 object. This object will be the main object that will allow you to interact with the BACnet network.

More than one BAC0 object can be created but only one connection by interface is supported.

Typically, we'll call this object 'bacnet' to illustrate that it represents the access point to the BACnet network. But you can call it like you want.

This object will be used to interact with the BACnet network. It will be used to discover devices, read and write properties, trend points, etc.

This object will also be used as a BACnet device itself, serving BACnet objects to the network.

To create a BAC0 object, you will need to use the start() function. This function will create the object and connect it to the network.

> **Note**
>
> Legacy BAC0 was available in 2 flavours : ilte and complete. This is not the case anymore. I have merged the two versions into one. All web services have been deprecated letting other softwares like Grafana or InfluxDB to take care of the trending features.

When creating the connection to the network, BAC0 needs to know the ip network of the interface on which it will work. It also needs to know the subnet mask (as BACnet operations often use broadcast messages).If you don't provide one, BAC0 will try to detect the interface for you.

> **Note**
>
> Legacy BAC0 have been tested with Pythonista (iOS) to a certain point. This is not the case for this version where I haven't tested this and have no plan to support it.

2.2.2 How to run async code

To run async code interactively (so you can explore the library or your BACnet network), you can use different ways. Running standalone scripts are somewhat different and we'll cover that later.

The first way is to use the REPL by calling : *python -m asyncio*. This will start the asyncio REPL and you will be able to run async code. This is the recommended way to run async code interactively.

Another way is to use a Jupyter Notebook. This is also a good way to run async code interactively. They can even run directly inside your browser or code editor using .ipynb files.

2.2.3 Define a bacnet application

Example:

```
import BAC0
bacnet = BAC0.start()
# or specify the IP you want to use / bacnet = BAC0.start(ip='192.168.1.10/24')
# by default, it will attempt an internet connection and use the network adapter
# connected to the internet.
# Specifying the network mask will allow the usage of a local broadcast address
# like 192.168.1.255 instead of the global broadcast address 255.255.255.255
# which could be blocked in some cases.
```

2.2.4 Dependencies and nice to hace features

BAC0 is a library that relies on several other libraries to function effectively. The main library is bacpypes3, a BACnet stack that facilitates interaction with BACnet devices. BAC0 serves as a wrapper around bacpypes3, simplifying these interactions.

In this new version, rich is used to enhance console output, making it more readable. While not required, it improves the user experience. Similarly, Pandas is utilized for handling historical data, easing the process of working with such data.

Additionally, python-dotenv is employed to load environment variables from a .env file, simplifying the management of these variables. For data storage, sqlite3 is used to maintain a local database when devices are disconnected, and InfluxDB is used for storing data in a time series database. Both are optional but provide convenient data storage solutions.

2.2.4.1 Asynchronous programming

BAC0 is based on the new bacpypes3 which highly relies on asyncio. This means that you will have to use the asyncio library to interact with BAC0. This is not a big deal as asyncio is now part of the standard library and is easy to use. But it brings some changes in the way you will write your code. Typically, all requests to read information on the network will be required to be awaited. This is done to allow the event loop to continue to run and not block the execution of the program. Some functions will not require to be awaited, for example, write requests for which we don't need to receive an message back.

Asynchronous programming brings some overhead in the way you will write your code. But it also brings a lot of advantages. For example, you can now write code that will be able to do multiple things at the same time. This is really useful when you want to read multiple points on the network at the same time. Your code will run faster.

2.2.4.2 Lite vs Complete vs connect vs start

This version of BAC0 present only one way to create a BAC0 object. The function *start()* is the preferred way to create a BAC0 object. This function will create a BAC0 object and connect it to the network. To maintain compatibility with previous versions, the functions *connect()* and *lite()* are still available. They are exact equivalent of *start()*.

2.2.4.2.1 Start

When you start BAC0, you will have a BAC0 object that will be able to interact with the BACnet network. This object will be able to discover devices, read and write properties, trend points, etc. But before doing so, you will need to know some details about the network you want to connect to. Using the same subnet is really important as BACnet relies on broadcast messages to communicate. If you don't know the subnet, BAC0 will try to find it for you. But it is always better to provide it. If you need to connect to a network that is not on the same subnet as the one you are on, you will need to provide the IP address of a BBMD (BACnet Broadcast Management Device) that will be able to route the messages to the network you want to connect to.

BAC0 can act as a BBMD itself and route messages to other networks. But in simple cases where you will only want to explore the network, being configured as a foreign device is enough.

Details about configuring BAC0 as a foreign device or a BBMD are available in the documentation.

To do so, use the syntax:

```
bacnet = BAC0.start(ip='xxx.xxx.xxx.xxx/mask')
```

> Device ID > > It's possible to define the device ID you want in your BAC0 instance by > using the *deviceId* argument *bacnet = BAC0.start(ip='xxx.xxx.xxx.xxx/mask', deviceId=1234).*

2.2.4.3 Use BAC0 on a different subnect (Foreign Device)

In some situations (like using BAC0 with a VPN using TUN) your BAC0 instance will run on a different subnet than the BACnet/IP network.

BAC0 support being used as a foreign device to cover those cases.

You must register to a BBMD (BACnet Broadcast Management Device) that will organize broadcast messages so they can be sent through diferent subnet and be available for BAC0.

To do so, use the syntax:

```
my_ip = '10.8.0.2/24'
bbmdIP = '192.168.1.2:47808'
bbmdTTL = 900
bacnet = BAC0.start(ip='xxx.xxx.xxx.xxx/mask', bbdmAddress=bbmdIP, bbmdTTL=bbmdTTL)
```

2.2.4.4 Discovering devices on a network

BACnet protocole relies on "whois" and "iam" messages to search and find devices. Typically, those are broadcast messages that are sent to the network so every device listening will be able to answer to whois requests by a iam request.

By default, BAC0 will use "local broadcast" whois message. This mean that in some situation, you will not see by default the global network. Local broadcast will not traverse subnets and won't propagate to MSTP network behind BACnet/IP-BACnet/MSTP router that are on the same subnet than BAC0.

This is done on purpose because using "global broadcast" by default will create a great amount of traffic on big BACnet network when all devices will send their "iam" response at the same time.

Instead, it is recommended to be careful and try to find devices on BACnet networks one at a time. For that though, you have to "already know" what is on your network. Which is not always the case. This is why BAC0 will still be able to issue global broadcast whois request if explicitly told to do so.

The recommended function to use is

```
bacnet.discover(networks=['listofnetworks'], limits=(0,4194303), global_broadcast=False)
# networks can be a list of integers, a simple integer, or 'known'
# By default global_broadcast is set to False
# By default, the limits are set to any device instance, user can choose to request only␣
↪a
# range of device instances (1000,1200) for instance
```

This function will trigger the whois function and get you results. It will also emit a special request named 'What-si-network-number' to try to learn the network number actually in use for BAC0. As this function have been added in the protocole 2008, it may not be available on all networks.

BAC0 will store all network number found in the property named *bacnet.known_network_numbers*. User can then use this list to work with discover and find everything on the network without issuing global broadcasts. To make a discover on known networks, use

```
bacnet.discover(networks='known')
```

Also, all found devices can be seen in the property *bacnet.discoveredDevices*. This list is filled with all the devices found when issuing whois requests.

BAC0 also provide a special functions to get a device table with details about the found devices. This function will try to read on the network for the manufacturer name, the object name, and other informations to present all the devices in a pandas dataframe. This is for presentation purposes and if you want to explore the network, I recommend using discover.

Devices dataframe

```
await bacnet.devices
```

..note::

WARNING. *await bacnet.devices* may in some circumstances, be a bad choice when you want to discover devices on a network. A lot of read requests are made to look for manufacturer, object name, etc and if a lot of devices are on the network, it is recommended to use whois() and start from there.

BAC0 also support the 'Who-Is-Router-To-Network' request so you can ask the network and you will see the address of the router for this particular BACnet network. The request 'Initialize-Router-Table' will be triggered on the reception of the 'I-Am-Router-To-Network' answer.

Once BAC0 will know which router leads to a network, the requests for the network inside the network will be sent directly to the router as unicast messages. For example

```
# if router for network 3 is 192.168.1.2
bacnet.whois('3:*')
# will send the request to 192.168.1.2, even if by default, a local broadcast would sent␣
↪the request
# to 192.168.1.255 (typically with a subnet 255.255.255.0 or /24)
```

2.2.4.5 Ping devices (monitoring feature)

BAC0 includes a way to ping constantly the devices that have been registered. This way, when devices go offline, BAC0 will disconnect them until they come back online. This feature can be disabled if required when declaring the network

```
bacnet = BAC0.start(ping=False)
```

By default, the feature is activated.

When reconnecting after being disconnected, a complete rebuild of the device is done. This way, if the device have changed (a download have been done and point list changed) new points will be available. Old one will not.

..note::
WARNING. When BAC0 disconnects a device, it will try to save the device to SQL.

2.2.4.6 Routing Table

BACnet communication trough different networks is made possible by the different routers creating "routes" between the subnet where BAC0 live and the other networks. When a network discovery is made by BAC0, informations about the detected routes will be saved (actually by the bacpypes stack itself) and for reference, BAC0 offers a way to extract the information

```
await bacnet.routing_table
```

This will return a dict with all the available information about the routes in this form :

await bacnet.routing_table Out[5]: {'192.168.211.3': Source Network: None | Address: 192.168.211.3 | Destination Networks: {303: 0} | Path: (1, 303)}

2.3 Read from network

To read from a BACnet device using the bacnet instance just created by the connection. You must know what you are trying to read though and some technical specificities of BACnet. Let's have a simple look at how things work in BACnet.

To read to a point, you need to create a request that will send some message to the network (directly to a controller (unicast) or at large (broadcast) with the object and property from the object you want to read from. The BACnet standard defines a lot of different objects. All objects provides some properties that we can read from. You can refer bacpypes source code (object.py) to get some examples.

For the sake of the explanation here, we'll take one common object : an analog value.

The object type is an **analogValue**. This object contains properties. Let's have a look to bacpypes definition of an AnalogValue

```
@register_object_type
class AnalogValueObject(Object):
    objectType = 'analogValue'
    _object_supports_cov = True

    properties = \
        [ ReadableProperty('presentValue', Real)
        , ReadableProperty('statusFlags', StatusFlags)
        , ReadableProperty('eventState', EventState)
```

(continues on next page)

(continued from previous page)

```
        , OptionalProperty('reliability', Reliability)
        , ReadableProperty('outOfService', Boolean)
        , ReadableProperty('units', EngineeringUnits)
        , OptionalProperty('minPresValue', Real)
        , OptionalProperty('maxPresValue', Real)
        , OptionalProperty('resolution', Real)
        , OptionalProperty('priorityArray', PriorityArray)
        , OptionalProperty('relinquishDefault', Real)
        , OptionalProperty('covIncrement', Real)
        , OptionalProperty('timeDelay', Unsigned)
        , OptionalProperty('notificationClass',  Unsigned)
        , OptionalProperty('highLimit', Real)
        , OptionalProperty('lowLimit', Real)
        , OptionalProperty('deadband', Real)
        , OptionalProperty('limitEnable', LimitEnable)
        , OptionalProperty('eventEnable', EventTransitionBits)
        , OptionalProperty('ackedTransitions', EventTransitionBits)
        , OptionalProperty('notifyType', NotifyType)
        , OptionalProperty('eventTimeStamps', ArrayOf(TimeStamp, 3))
        , OptionalProperty('eventMessageTexts', ArrayOf(CharacterString, 3))
        , OptionalProperty('eventMessageTextsConfig', ArrayOf(CharacterString, 3))
        , OptionalProperty('eventDetectionEnable', Boolean)
        , OptionalProperty('eventAlgorithmInhibitRef', ObjectPropertyReference)
        , OptionalProperty('eventAlgorithmInhibit', Boolean)
        , OptionalProperty('timeDelayNormal', Unsigned)
        , OptionalProperty('reliabilityEvaluationInhibit', Boolean)
        ]
```

Readable properties are mandatory and all BACnet device must implement AnalogValue with those properties. Optional properties, may or may not be available, depending on the choices the manufacturer made.

With BAC0, there is two different kind of requests we can use to read from the network

- read
- readMultiple

Read will be used to read only one (1) property. readMultiple will be used to read multiple properties. The number here is not defined. Many elements will have an impact on the number of properties you can retrieve using readMultiple. Is the device an MSTP one or a IP one. Does the device support segmentation ? Etc. Many details that could prevent you from using readMultiple with very big requests. Usually, when discovering a device points, BAC0 will use readMultiple and will use chunks of 25 properties. It's up to you to decide how many properties you'll read.

So, to read from BACnet. The request will contains the address of the device from which we want to read (example '2:5'). Then the object type (analogValue), the instance number (the object "address" or "register"... let's pretend it's 1) and the property from the object we want to read (typically the 'presentValue').

2.3.1 Read examples

Once you know the device you need to read from, you can use

```
bacnet.read('address object object_instance property')
```

Read property multiple can also be used

```
bacnet.readMultiple('address object object_instance property_1 property_2') #or
bacnet.readMultiple('address object object_instance all')
```

2.3.2 Read multiple

Using simple read is a costly way of retrieving data. If you need to read a lot of data from a controller, and this controller supports read multiple, you should use that feature.

When defining *BAC0.devices*, all polling requests will use readMultiple to retrive the information on the network.

There is actually two way of defining a read multiple request. The first one inherit from bacpypes console examples and is based on a string composed from a list of properties to be read on the network. This is the example I showed previously.

Recently, a more flexible way of creating those requests have been added using a dict to create the requests. The results are then provided as a dict for clarity. Because the old way just give all the result in order of the request, which can lead to some errors and is very hard to interact with on the REPL.

The *request_dict* must be created like this

```
_rpm = {'address': '303:9',
        'objects': {
            'analogInput:1094': ['objectName', 'presentValue', 'statusFlags', 'units',
↪'description'],
            'analogValue:4410': ['objectName', 'presentValue', 'statusFlags', 'units',
↪'description']
            }
        }
```

If an array index needs to be used, the following syntax can be used in the property name

```
# Array index 1 of propertyName
'propertyName@idx:1'
```

This dict must be used with the already exsiting function *bacnet.readMultiple()* and passed via the argument named **request_dict**.

```
bacnet.readMultiple('303:9', request_dict=_rpm)
```

The result will be a dict containing all the information requested.

```
# result of request
{
    ('analogInput', 1094): [
        ('objectName', 'DA-VP'),
        ('presentValue', 4.233697891235352),
        ('statusFlags', [0, 0, 0, 0]),
```

(continues on next page)

(continued from previous page)

```
        ('units', 'pascals'),
        ('description', 'Discharge Air Velocity Pressure')
        ],
    ('analogValue', 4410): [
        ('objectName', 'SAFLOW-ABSEFFORT'),
        ('presentValue', 0.005016503389924765),
        ('statusFlags', [0, 0, 1, 0]),
        ('units', 'percent'),
        ('description', '')
        ]
}
```

2.3.3 Write to property

To write to a single property

```
bacnet.write('address object object_instance property value - priority')
```

2.3.4 Write to multiple properties

Write property multiple is also implemented. You will need to build a list for your requets

```
r = ['analogValue 1 presentValue 100','analogValue 2 presentValue 100','analogValue 3␣
↪presentValue 100 - 8','@obj_142 1 @prop_1042 True']
bacnet.writeMultiple(addr='2:5',args=r,vendor_id=842)
```

..note::
WARNING. See the section on Proprietary objects and properties for details about vendor_id and @obj_142.

2.4 Write to network

Write is very similar to read in term of concept. You typically want to write a value to a property which is part of an object.

So to write to the *presentValue* of an analogValue (if we keep the same object than in the read chapter) will need you to tell BAC0 to which address you want to write, which object, at what instance and what property, just like when you read.

But you also need to provide supplemental information :

- what value you want to write
- what priority

2.4.1 priority

In BACnet, object to which we can write often provide what is called the *priorityArray*. This array contains 16 levels to which we can write (1 being the highest priority).

Typical usage of priority is :

> 1 Manual-Life Safety 2 Automatic-Life Safety 3 Available 4 Available 5 Critical Equipment Control 6 Minimum On/Off 7 Available 8 Manual Operator (Override) 9 Available 10 Available (Typical Control from a Supervisor) 11 Available 12 Available 13 Available 14 Available 15 Available (Schedule) 16 Available

2.4.2 Write to a simple property

To write to a single property

```
bacnet.write('address object object_instance property value - priority')
```

2.4.3 Write to multiple properties

Write property multiple is also implemented. You will need to build a list for your requets

```
r = ['analogValue 1 presentValue 100','analogValue 2 presentValue 100','analogValue 3␣
↪presentValue 100 - 8','@obj_142 1 @prop_1042 True']
bacnet.writeMultiple(addr='2:5',args=r,vendor_id=842)
```

..note::
WARNING. See the section on Proprietary objects and properties for details about vendor_id and @obj_142.

2.5 Time Sync

You can use BAC0 to send time synchronisation requests to the network

```
bacnet.time_sync()
# or
bacnet.time_sync('2:5') # <- Providing an address
```

BAC0 will not accept requests from other devices.

2.6 How to define a device and interact with points

2.6.1 Define a controller

Once the bacnet variable is created, you can define devices.

Example:

```
import BAC0
bacnet = BAC0.start()
# or specify the IP you want to use / bacnet = BAC0.start(ip='192.168.1.10/24')
# by default, it will attempt an internet connection and use the network adapter
```

(continues on next page)

(continued from previous page)

```
# connected to the internet.
# Specifying the network mask will allow the usage of a local broadcast address
# like 192.168.1.255 instead of the global broadcast address 255.255.255.255
# which could be blocked in some cases.

# Query and display the list of devices seen on the network
bacnet.discover()
bacnet.devices

# Define a controller (this one is on MSTP #3, MAC addr 4, device ID 5504)
mycontroller = BAC0.device('3:4', 5504, bacnet)

# Get the list of "registered" devices
bacnet.registered_devices
```

2.6.1.1 Some caveats

2.6.1.1.1 Segmentation

Some devices do not support segmentation. BAC0 will try to detect that and will not allow "read property multiple" to be used. But it is sometimes better to speciy to BAC0 that the device doesn't support segmentation.

To do so, use the parameter:

```
my_old_device = BAC0.start('3:4', 5504, bacnet, segmentation_supported=False)
```

2.6.1.1.2 Object List

By default, BAC0 will read the object list from the controller and define every points found inside the device as points. This behaviour may not be optimal in all use cases. BAC0 allows you to provide a custom object list when creating the device.

To do so, use this syntax:

```
# Define your own list
my_obj_list = [('file', 1),
               ('analogInput', 2),
               ('analogInput', 3),
               ('analogInput', 5),
               ('analogInput', 4),
               ('analogInput', 0),
               ('analogInput', 1)]

# Provide it as an argument
fx = BAC0.device('2:5',5,bacnet, object_list = my_obj_list)
```

2.6.2 Look for points in controller

Example:

```
mycontroller.points
```

2.6.3 Read the value of a point

To read a point, simply ask for it using bracket syntax:

```
mycontroller['point_name']
```

2.6.4 Writing to Points

2.6.4.1 Simple write

If point is a value:

- analogValue (AV)
- binaryValue (BV)
- multistateValue (MV)

You can change its value with a simple assignment. BAC0 will write the value to the object's **presentValue** at the default priority.:

```
mycontroller['point_name'] = 23
```

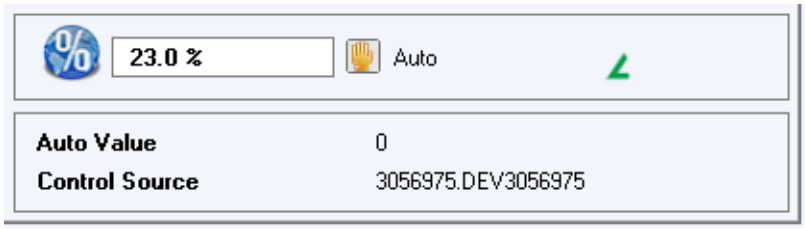


Fig. 1: *Example from Delta Controls OWS Workstation*

2.6.4.2 Write to an Output (Override)

If the point is an output:

- analogOutput (AO)
- binaryOutput (BO)
- multistateOutput (MO)

You can change its value with a simple assignment. BAC0 will write the value to the object's **presentValue** (a.k.a override it) at priority 8 (Manual Operator).:

```
mycontroller['outputName'] = 45
```

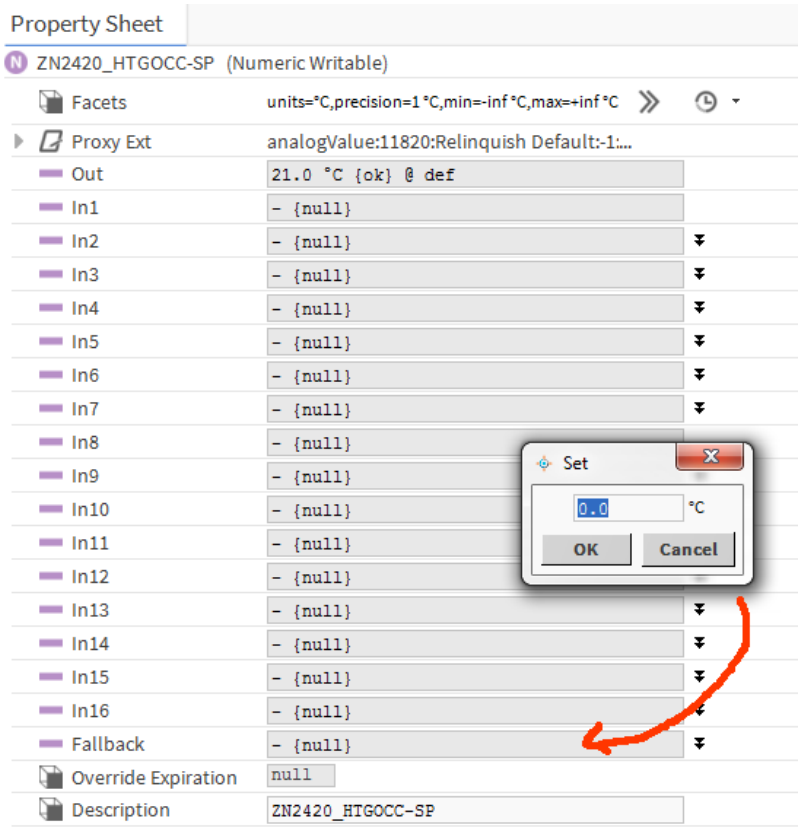


Fig. 2: *Example from Niagara 4 station*

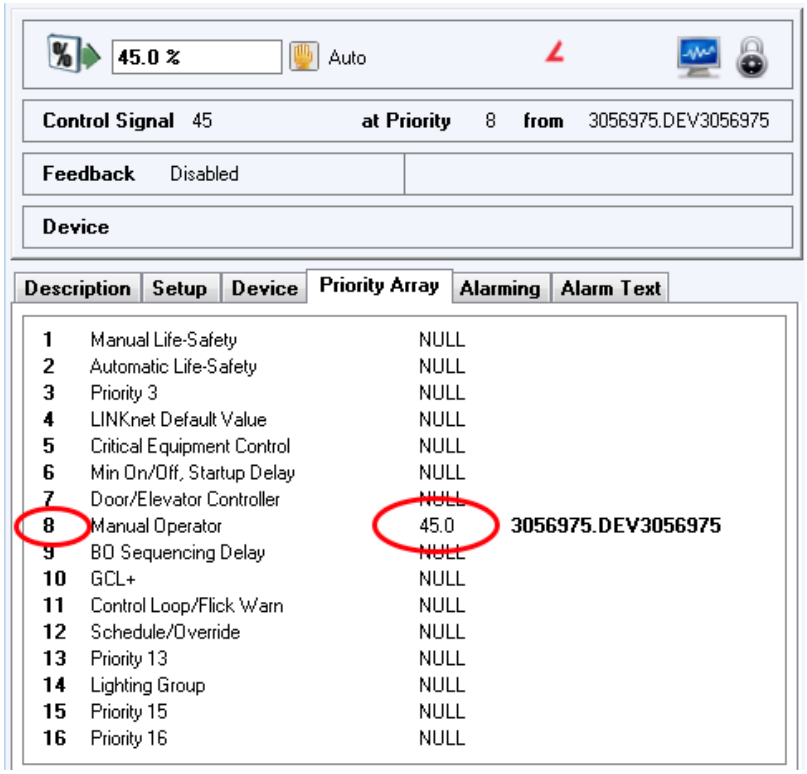


Fig. 3: *Example from Delta Controls OWS Workstation*

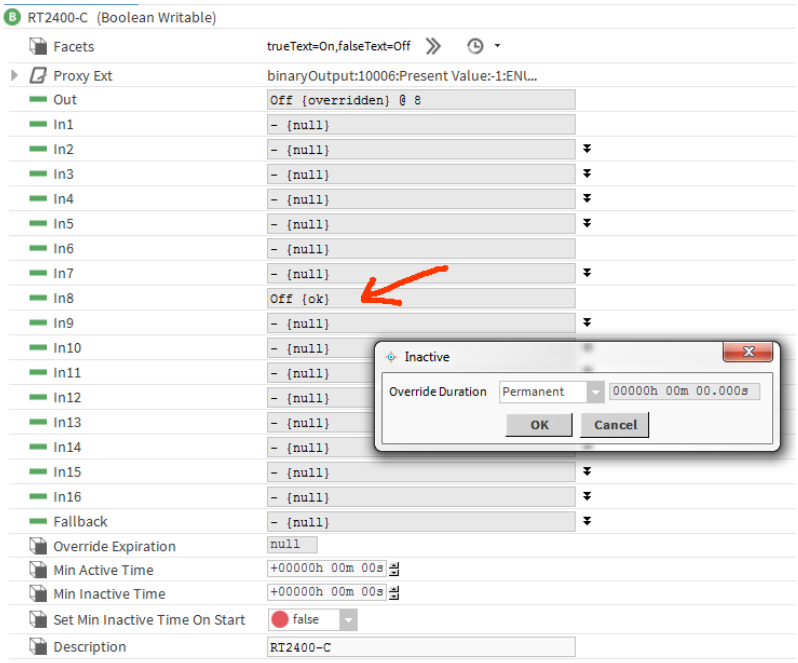


Fig. 4: *Example from Niagara 4 station*

2.6.4.3 Write to an Input (simulate)

If the point is an input:

- analogInput (AI)
- binaryOutput (BO)
- multistateOutput (MO)

You can change its value with a simple assigment, thus overriding any external value it is reading and simulating a different sensor reading. The override occurs because BAC0 sets the point's **out_of_service** (On) and then writes to the point's **presentValue**.

mycontroller['inputName'] = <simulated value>

mycontroller['Temperature'] = 23.5 # overiding actual reading of 18.8 C

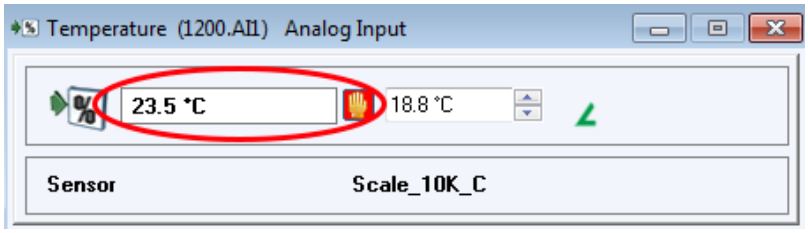


Fig. 5: *Example from Delta Controls OWS Workstation*

In a Niagara station, you would need to create a new point using the "out_of_service" property, then set this point to True. Then you would need to create (if not already done) a point writable to the present value property and write to it. No screenshot available.

2.6.4.4 Releasing an Input simulation or Output override

To return control of an Input or Output back to the controller, it needs to be released. Releasing a point returns it automatic control. This is done with an assignment to 'auto'.:

```
mycontroller['pointToRelease'] = 'auto'
```

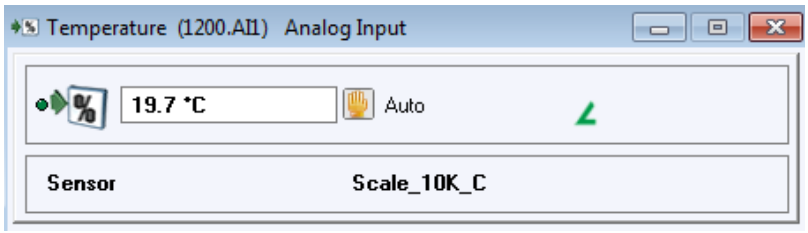


Fig. 6: *Example from Delta Controls OWS Workstation*

In a Niagara station, you would need to create a new point using the "out_of_service" property, then set this point to False. No screenshot available.

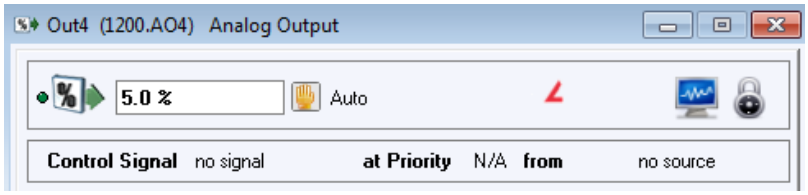


Fig. 7: *Example from Delta Controls OWS Workstation*

2.6.4.5 Setting a Relinquish_Default

When a point (with a priority array) is released of all override commands, it takes on the value of its **Relinquish_Default**. [BACnet clause 12.4.12] If you wish to set this default value, you may with this command:

```
await mycontroller['pointToChange'].default(<value>)
await mycontroller['Output'].default(75)
```

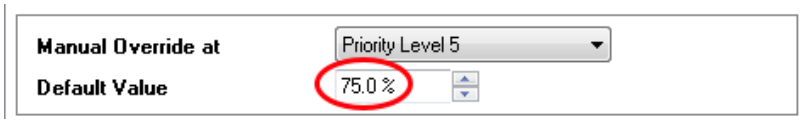


Fig. 8: *Example from Delta Controls OWS Workstation*

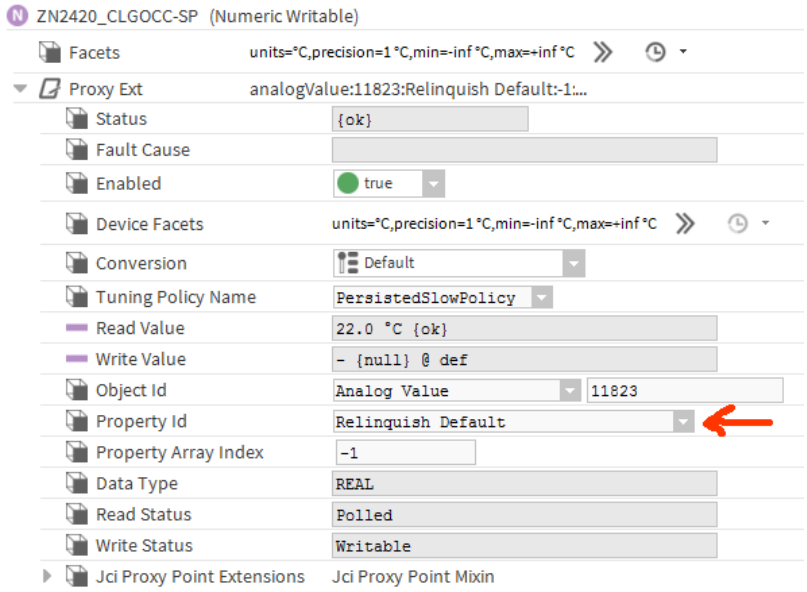


Fig. 9: *Example from Niagara 4 station*

2.6.4.6 BACnet properties

BAC0 defines its own "image" of a controller. All points inside a *BAC0.device* are Python objects with which we can interact. If you want to access native BACnet objects and properties there are functions you can use.

2.6.4.6.1 Read all device properties

You can retrieve the list of device properties using:

```
device.bacnet_properties
# will return a cached version by default. If things have changed, you can refresh using.
device.update_bacnet_properties()
```

Often, in this list, you will see proprietary properties added by the manufacturer. They can be recognize by their name, an integer.

2.6.4.6.2 Read Property

You can read simple properties using

```
prop = ('device',100,'objectName')
await device.read_property(prop)
# this will return the object name
prop = ('analogInput',1,'priorityArray')
await device.read_property(prop)
# this will return the priority array of AI1
```

2.6.4.6.3 Write property

You can write to a property using

```
prop = ('analogValue',1,'presentValue')
await device.write_property(prop,value=98,priority=7)
```

2.6.4.6.4 Write description

The **write_property** method will not work to update a description if it contains a space.

Instead, use **update_description** against a point:

```
await device['AI_3'].update_description('Hello, World!')
```

You can then read the description back, as a property:

```
await device['AI_3'].read_property('description')
```

or going back to the device:

```
await device.read_property(('analogInput',3,'description'))
```

2.7 Proprietary Objects

Some manufacturers provide special variables inside their controllers in the form of proprietary objects or expand some objects with proprietary properties. BAC0 supports the creation of those objects but some work is needed on your side to register them.

In fact, you will need to know what you are looking for when dealing with proprietary objects or properties. Should you write to them or make them read only ? What type should you declare ?

Once you know the information, you are ready to make your implementation.

The actual BAC0 implementation allow the user to be able to read proprietary objects or proprietary properties without defining a special class. This is done using a special syntax that will inform BAC0 of the nature or the read.

Why ? Bacpypes requests (in BAC0) are made sequencially using well-known property names and address. When dealing with proprietary objects or properties, names and addresses are numbers. This is somewhat hard to detect if the request contains an error, is malformed or contains a proprietary thing in it. The new syntax will tell BAC0 that we need to read a proprietary object or property.

If you need to read an object named "142", you will tell BAC0 to read *@obj_142* If you need to read a property named 1032, you will tell BAC0 to read *@prop_1032*

This way, you could build a request this way :

> await bacnet.read('2:5 @obj_142 1 @prop_1032') # or await bacnet.readMultiple('2:5 @obj_142 1 objectName @prop_1032')

2.7.1 Writing to proprietary properties

If you need to write to the property, things are a litlle more complicated. For example, JCI TEC3000 have a variable that needs to be written to so the thermostat know that the supervisor is active, a condition to use network schedule (if not, switch to internal schedule).

If you try this :

> await bacnet.write('2000:10 device 5010 3653 True')

You'll get :

> TypeError: issubclass() arg 1 must be a class

This is because BAC0 doesn't know how to encode the value to write. You will need to define a class, register it so BAC0 knows how to encore the value and most importantly, you will need to provide the *vendor_id* to the write function so BAC0 will know which class to use. Because 2 different vendors could potentially use the same "number" for a proprietary object or property with different type.

2.7.1.1 How to implement [DEPRECATED - MUST BE UPDATED]

BAC0 will allow dynamic creation of the classes needed to read and write to those special variables. To do so, a special dictionary need to be declared in this form ::

```
name = {
    "name": "Class_Name",
    "vendor_id": integer,
    "objectType": "type",
    "bacpypes_type": Object,
    "properties": {
```

(continues on next page)

(continued from previous page)

```
        "NameOfProprietaryProp": {"obj_id": 1110, "datatype": Boolean, "mutable": True},
    },
}

# name : Name of the class to be created
# vendor_id : the manufacturer of the device
# objectType : see bacpypes.object for reference (ex. 'device')
# bacpypes_type : base class to instanciate (ex. BinaryValueObject)
# properties : list of proprietary properties to add
#     name of the property (for reference)
#     obj_id : instance of the property, usually an integer
#     datatype : the kind of data for this property. Refer to `bacpypes.primitivedata` or␣
↪`bacpypes.constructeddata`
#     mutable : true = writable, default to false
```

Once the dictionary is completed, you need to call the spceial function *create_proprietaryobject*. This function will dynamically create the class and register it with bacpypes so you will be able to read and write to the object.

To access the information (for now), you will use this syntax

```
# Suppose an MSTP controller at address 2:5, device instance 5003
# Vendor being Servisys (ID = 842)
# Proprietary property added to the device object with object ID 1234
bacnet.read('2:5 device 5003 1234', vendor_id=842)
```

If you want to look at the object registration, you can use this

```
from bacpypes.object import registered_object_types
registered_object_types
```

It is a dictionary containing all the registered type in use. As you can see, the majority of the registration use vendor_id 0 which is the default. But if you register something for another vendor_id, you will see a new dictionary entry. Using the special *bacnet.read* argument "vendor_id" will then inform bacpypes that we want to use the special object definition for this particular vendor.

Note

BAC0 will automatically register known proprietary classes at startup. See BAC0.core.proprietary_objects for details.

2.7.1.2 Proprietary objects [DEPRECATED - MUST BE UPDATED]

Proprietary object can be accessed using

```
# Let say device '2:5' have object (140,1)
bacnet.read('2:5 140 1 objectName')
```

As they are proprietary objects, you will have to know what you are looking for. Typically, the properties *objectName*, *objectIdentifier*, will be available. But you will often see proprietary properties attached to those objects. See next section.

To read all properties from an object, if implemented, one can use

```
bacnet.readMultiple('2:5 140 1 all')
```

BAC0 will do its best to give you a complete list.

> **Note**
>
> Please note that arrays under proprietary objects are not implemented yet. Also, context tags objects are not detected automatically. You will need to build the object class to interact with those objects. See next section.

2.7.1.3 Proprietary Property [DEPRECATED - MUST BE UPDATED]

One common case I'm aware of is the addition of proprietary properties to the DeviceObject of a device. Those properties may, for example, give the CPU rate or memory usage of the controllers. On the TEC3000 (JCI), there is a "SupervisorOnline" property needed to be written to, allowing the BAS schedule to work.

To declare those properties, we need to extend the base object (the DeviceObject in this case) pointing this declaration to the vendor ID so bacpypes will know where to look.

The following code is part of BAC0.core.proprietary_objects.jci and define proprietary properties added to the device object for JCI devices. Note that as there are multiple proprietary properties, we need to declare them all in the same new class (the example presents 2 new properties).

```
#
#   Proprietary Objects and their attributes
#

JCIDeviceObject = {
    "name": "JCI_DeviceObject",
    "vendor_id": 5,
    "objectType": "device",
    "bacpypes_type": DeviceObject,
    "properties": {
        "SupervisorOnline": {"obj_id": 3653, "datatype": Boolean, "mutable": True},
        "Model": {"obj_id": 1320, "datatype": CharacterString, "mutable": False},
    },
}
```

This will allow us to interact with them after registration

```
from BAC0.core.proprietary_objects.jci import JCIDeviceObject
from BAC0.core.proprietary_objects.object import create_proprietaryobject
create_proprietaryobject(**JCIDeviceObject)

# Read model of TEC
bacnet.read('2:5 device 5005 1320', vendor_id=5)
# Write to supervisor Online
bacnet.write('2:5 device 5005 3653 true',vendor_id=5)
```

> **Note**
>
> In future version it will be able to define special device and attach some proprietary objects to them so tec['SupOnline'] would work…

2.7.2 Vendor Context for Read and Write [DEPRECATED - MUST BE UPDATED]

In *BAC0.device*, the vendor_id context will be provided to the stack automatically. This mean that if a device is created and there is a extended implementation of an object (JCIDeviceObject for example) BAC0 will recognize the proprietary object by default, without having the need to explicitly define the vendor_id in the request

```
instance_number = 1000
prop_id = 1320
device.read_property(('device',instance_number, prop_id))
```

will work.

Also, proprietary objects and properties classes are defined at startup so it is not necessary to explicitly register them.

2.7.3 Can proprietary objects be addded to a BAC0.device points

Actually not, because of the way "points" are defined in BAC0. If you look at *BAC0.core.devices.Points.Point* you will see that the notion of point is oriented differently than a BACnet object. Properties are a set of informations useful for BAC0 itself but are not "strictly" BACnet properties. The value of a point will always be the *presentValue* of the BACnet object. In the context of proprietary objects, this can't fit.

There are no "standard" way to create a proprietary object. Beside the fact that objectName, objectType and objectIdentifier must be provided, everything else is custom.

For this reason, proprietary objects must be dealt outside of the scope of a device, especially in the context of writing to them.

2.7.4 How to implement readMultiple with proprietary objects and properties

It is possible to create read property multiple requests with them, using the syntax *@obj_* and *@prop_*. So for now, you will be able to create a request yourself for one device at a time by chaining properties you want to read :

> bacnet.readMultiple('2000:31 device 5012 @prop_3653 analogInput 1106 presentValue units')

2.7.5 How to find proprietary objects and properties

In BAC0, for a device or a point, you can use :

> device.bacnet_properties # or point.bacnet_properties

This will list *all* properties in the object. (equivalent of *bacnet.readMultiple('addr object id all')*)

2.8 Histories in BAC0

Histories in BAC0 were first introduced as the result of every reading the software made on each points, to keep trace of what was going on. It is different from the BACnet TrendLog object that may be configured in a device to keep history records in the memory of the controller.

Everytime a value is read in BAC0, the value will be stored in memory as what will be called from here : history.

TrendLog will also being accessible but for now, let's focus on BAC0's histories.

BAC0 uses the Python Data Analysis library **pandas** [http://pandas.pydata.org/] to maintain histories of point values over time. All points are saved by BAC0 in a **pandas** Series every 10 seconds (by default). This means you will automatically have historical data from the moment you connect to a BACnet device.

Access the contents of a point's history is very simple.:

```
controller['pointName'].history
```

Example

```
controller['Temperature'].history
2017-03-30 12:50:46.514947    19.632507
2017-03-30 12:50:56.932325    19.632507
2017-03-30 12:51:07.336394    19.632507
2017-03-30 12:51:17.705131    19.632507
2017-03-30 12:51:28.111724    19.632507
2017-03-30 12:51:38.497451    19.632507
2017-03-30 12:51:48.874454    19.632507
2017-03-30 12:51:59.254916    19.632507
2017-03-30 12:52:09.757253    19.536366
2017-03-30 12:52:20.204171    19.536366
2017-03-30 12:52:30.593838    19.536366
2017-03-30 12:52:40.421532    19.536366
dtype: float64
```

Note

pandas is an extensive data analysis tool, with a vast array of data manipulation operators. Exploring these is beyond the scope of this documentation. Instead we refer you to this cheat sheet [https://github.com/pandas-dev/pandas/blob/master/doc/cheatsheet/Pandas_Cheat_Sheet.pdf] and the pandas website [http://pandas.pydata.org/].

2.8.1 History Size

By default, BAC0 provides an unlimited history_size per points (number of records). But it could be useful in certain cases when the script will run for a long period of time and you want to keep control over memory

```
dev = BAC0.device('2:4',4,bacnet,history_size=2)
# or after...
dev.update_history_size(100)
# or just on one point :
dev['point'].properties.history_size = 30
```

2.8.2 Resampling data

One common task associated with point histories is preparing it for use with other tools. This usually involves (as a first step) changing the frequency of the data samples - called **resampling** in pandas terminology.

Since the point histories are standard pandas data structures (DataFrames, and Series), you can manipulate the data with pandas operators, as follows.:

```
# code snipet showing use of pandas operations on a BAC0 point history.

# Resample (consider the mean over a period of 1 min)
tempPieces = {
        '102_ZN-T' : local102['ZN-T'].history.resample('1min'),
        '102_ZN-SP' : local102['ZN-SP'].history.resample('1min'),
        '104_ZN-T' : local104['ZN-T'].history.resample('1min'),
        '104_ZN-SP' : local104['ZN-SP'].history.resample('1min'),
        '105_ZN-T' : local105['ZN-T'].history.resample('1min'),
        '105_ZN-SP' : local105['ZN-SP'].history.resample('1min'),
        '106_ZN-T' : local106['ZN-T'].history.resample('1min'),
        '106_ZN-SP' : local106['ZN-SP'].history.resample('1min'),
        '109_ZN-T' : local109['ZN-T'].history.resample('1min'),
        '109_ZN-SP' : local109['ZN-SP'].history.resample('1min'),
        '110_ZN-T' : local110['ZN-T'].history.resample('1min'),
        '110_ZN-SP' : local110['ZN-SP'].history.resample('1min'),
       }
# Remove any NaN values
temp_pieces = pd.DataFrame(tempPieces).fillna(method = 'ffill').fillna(method = 'bfill')

# Create a new column in the DataFrame which is the error between setpoint and␣
↪temperature
temp_pieces['Erreur_102'] = temp_pieces['102_ZN-T'] - temp_pieces['102_ZN-SP']
temp_pieces['Erreur_104'] = temp_pieces['104_ZN-T'] - temp_pieces['104_ZN-SP']
temp_pieces['Erreur_105'] = temp_pieces['105_ZN-T'] - temp_pieces['105_ZN-SP']
temp_pieces['Erreur_106'] = temp_pieces['106_ZN-T'] - temp_pieces['106_ZN-SP']
temp_pieces['Erreur_109'] = temp_pieces['109_ZN-T'] - temp_pieces['109_ZN-SP']
temp_pieces['Erreur_110'] = temp_pieces['110_ZN-T'] - temp_pieces['110_ZN-SP']

# Create a new dataframe from results and show some statistics
temp_erreurs = temp_pieces[['Erreur_102', 'Erreur_104', 'Erreur_105', 'Erreur_106',
↪'Erreur_109', 'Erreur_110']]
temp_erreurs.describe()
```

2.9 TrendLog

BACnet TrendLog is an object that a controller may implement. Once configured, it'll keep in memory a certain number of records for one particular point. Often though, as the controller memory may be limited, we'll have access to a limited number of records and the interval between each record may have been configured long enough to optimize the time frame coverage of the records. For example, taking records every 10 minutes instead of every 2 minutes will give 5 times longer time interval. Using a total of 2000 number of records, it will represent almost 14 days (10 min) vs less than 3 days (2 min).

BAC0 supports the reading of TrendLog objects and will convert the records to pandas Series if possible. This will allow to use **pandas** syntax over histories and make analysis easier for the user.

TrendLog objects have also been made compatible with the format required to be added as Bokeh Chart in the web interface.:

```
# Manually create a TrendLog
import BAC0
bacnet = BAC0.connect()
device = BAC0.device('2:5',5,bacnet)

# Given a TrendLog object at address 1 for device
trend = BAC0.TrendLog(1,device)

# Retrieve pandas serie
trend.history

# Adding this object to live trends
trend.chart()
```

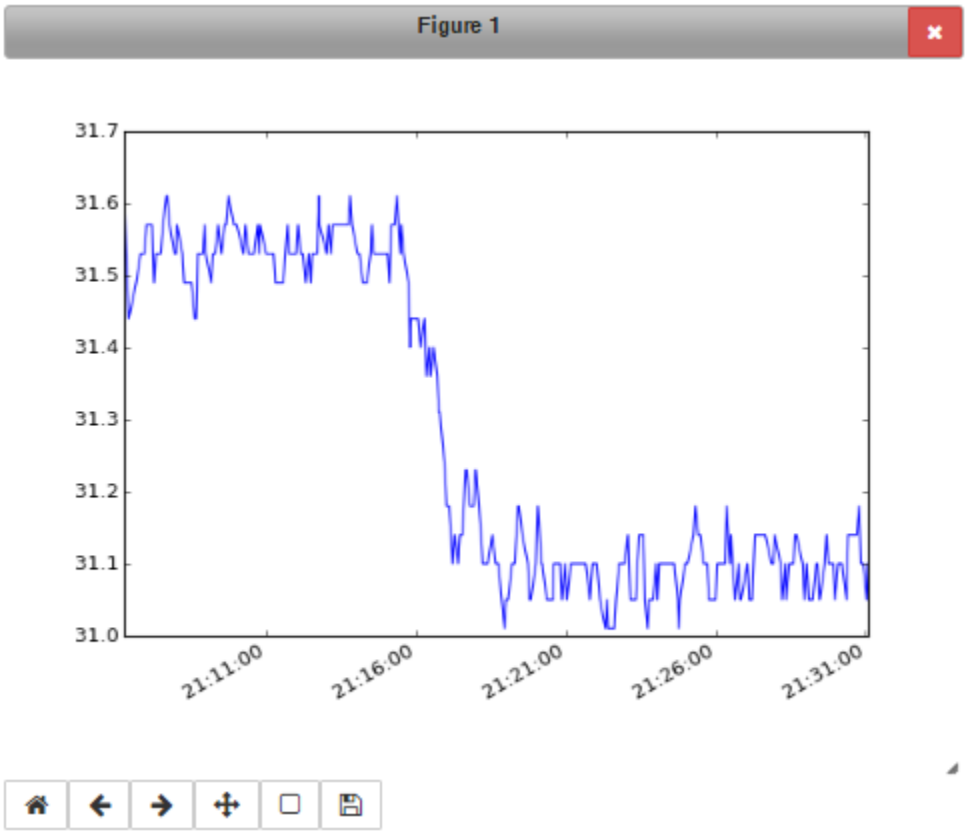
2.10 Trends

Trending is a nice feature when you want to see how a points value changed over time. This is only possible using matplotlib directly in Jupyter. And also in the Web Interface using Bokeh [http://bokeh.pydata.org/en/latest/] which brings a complete set of wonderful features for visualizing point histories (a.k.a. trends). The best feature of all - the ability to see Live Trends of your data as it occurs.

2.10.1 Matplotlib

Matplotlib is a well known data plotting library for Python. As BAC0's historical point data are pandas Series and DataFrames, it's possible to use Matplotlib with BAC0. i.e. Showing a chart using matplotlib:

```
%matplotlib notebook
# or matplotlib inline for a basic interface
controller['nvoAI1'].history.plot()
```



2.10.2 Seaborn

Seaborn is a library built over Matplotlib that extends the possibilities of creating statistical trends of your data. I strongly suggest you have a look to this library.

2.10.3 Bokeh

Bokeh is a Python interactive visualization library targeting modern web browsers for presentation. Its goal is to provide elegant, concise graphics, with high-performance interactivity over very large or streaming datasets. Bokeh can help anyone who would like to quickly create interactive plots, dashboards, and data applications.

Note

BAC0 trending features use Bokeh when running in "complete" mode. This requires the user to have some libraries installed :

- bokeh
- flask
- flask-bootstrap
- pandas
- numpy

> **Note**
>
> Running in "complete" mode may be hard to accomplish if you are running BAC0 on a Raspberry Pi. If doing so, I strongly recommend using the package berryconda which will install everything you need on the RPi to use Pandas, numpy... already compiled for the RPi.
>
> A simple call for "conda install bokeh" will install the package.

2.10.4 A web interface

To simplify the usage of the live trending feature, BAC0 implements a Web Server (running with Flask). Connect to http://localhost:8111 and you will get access to a Dashboard and the Trends page.

Internally, BAC0 will run two servers (flask and a bokeh server) that will handle the connection to the web interface and provide the web page with a live trend of the charts that have been sent to the interface.

2.10.5 Add/Remove plots to Bokeh

At first, the web page will be empty and no trend will appear. The user needs to specify which points must be trended. Points to trend are added to a list monitored by the "network" object. This will allow to add trends coming from multiple controllers easily

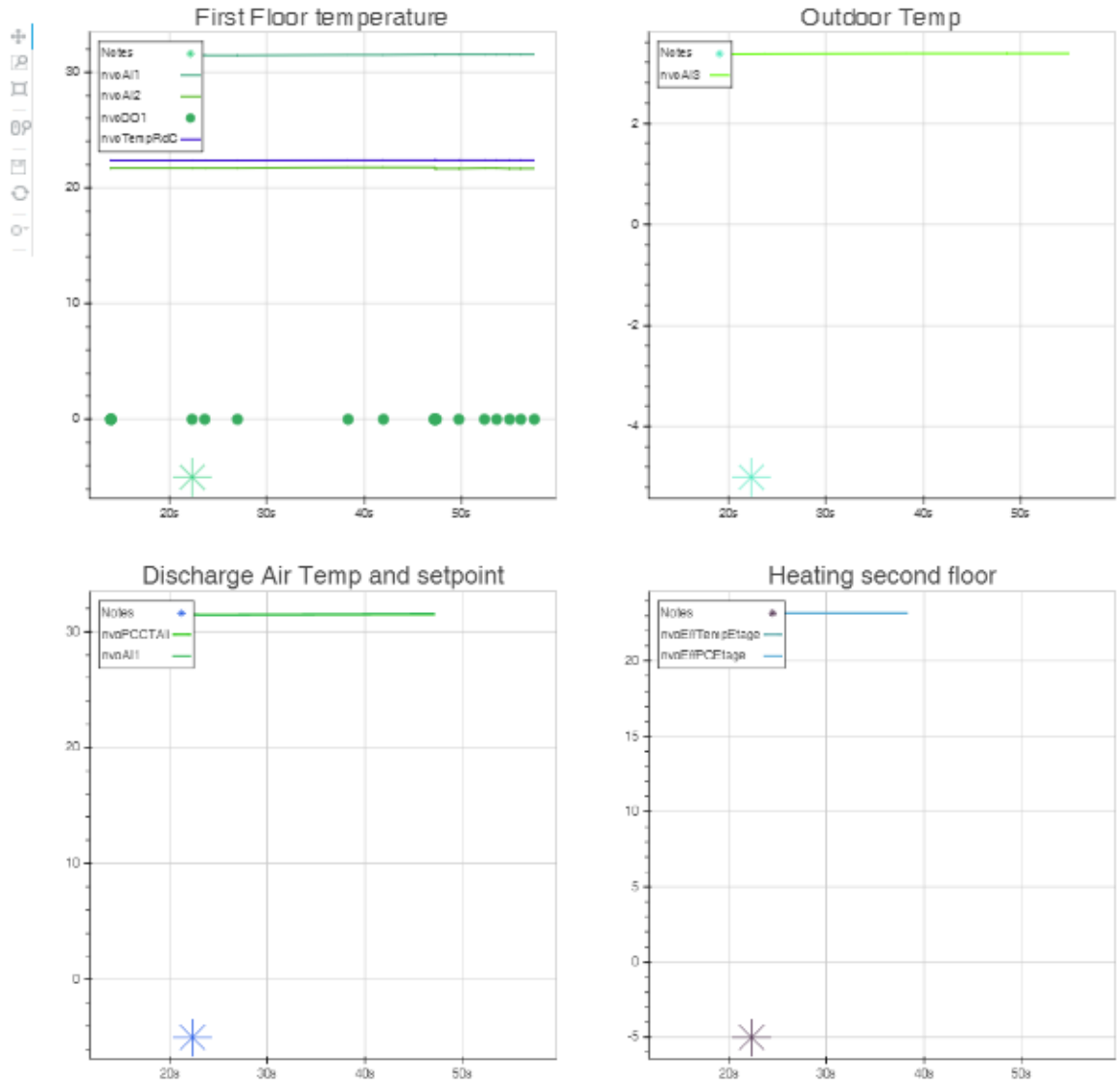
```
#each point can be added
controller['nvoAI1'].chart()

#or we can add them using the "network" object
bacnet.add_chart(controller['nvoAI1'])

# TrendLog object can also be added
trendlog_object.chart()
```

The list of trended points can be retrieve

```
bacnet.trends
#will give a list of all points added
```

To remove points

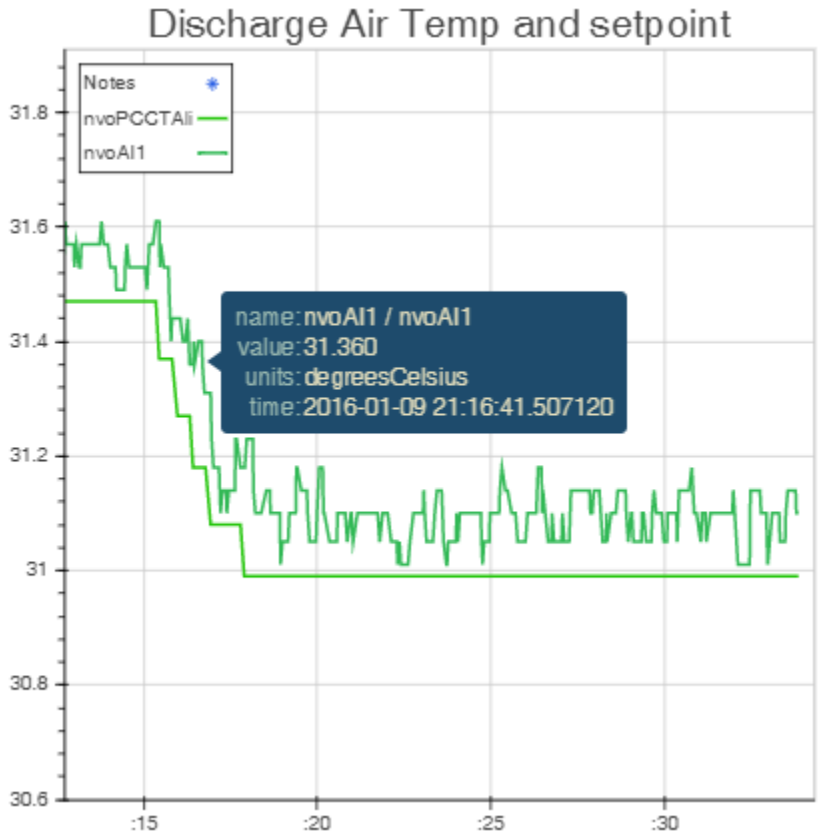
```
#on the point directly
controller['nvoAI1'].chart(remove=True)
bacnet.remove_chart(controller['nvoAI1'])
```



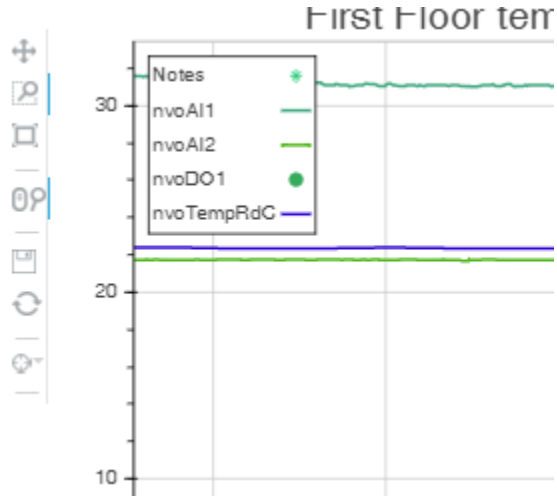
2.10.6 Bokeh Features

Bokeh has an extensive set of features. Exploring them is beyond the scope of this documentation. Instead you may discover them yourself at [http://www.bokehplots.com]. A couple of its features are highlighted below.

Hover tool:



And a lot of other options like pan, box zoom, mouse wheel zoom, save, etc…:



By default, x-axis will be a timeseries and will be linked between trends. So if you span one, or zoom one, the other plots will follow, giving you the eaxct same x-axis for every plots.

2.10.7 Bokeh Demo

Here is a working demo of Bokeh. It's taken from a real life test. You can use all the features (zoom, pan, etc.) Please note that the hover suffers from a little bug in this "saved" version of the trends... Working to solve this.

2.11 Schedules in BAC0

Schedules object in BAC0 are supported by using two specific functions

```
bacnet.read_weeklySchedule(address, instance)
# and
bacnet.write_weeklySchedule(address, instance, schedule)
```

This is required by the complexity of the object itself which is composed of multiple elements.

First, as you notice, actually, BAC0 support the "weeklySchedule" which is a property of the bacnet object ScheduleObject. The exceptionSchedule is not yet supported. Neither the calendar.

The weeklySchedule property is generally used locally inside the controller and is often synchronized from a supervisory controller if required.

weeklySchedule are made of 7 DailySchedules. Thoses schedules are made from lists of events written as TimeValues (a time of day, a value to be in).

This level of nesting would be very hard to write as a string to be passed to *bacnet.write* so this is why we provide 2 specific functions.

2.11.1 Python representation

One challenge in BAC0 is finding a good way to represent a BACnet object in the terminal. With schedules, the challenge was the quantity of elements important to understand what is going on with the schedule, what are the events, what is the actual value, the priorityForWriting, etc... Important informations when you interact with a controller. But also, we needed a simple format allowing easy editing to be written to the controlle.

The dict was simple enough to hold all the information and the chosen format is

```
schedule_example_multistate = {
    "states": {"Occupied": 1, "UnOccupied": 2, "Standby": 3, "Not Set": 4},
    "week": {
        "monday": [("1:00", "Occupied"), ("17:00", "UnOccupied")],
        "tuesday": [("2:00", "Occupied"), ("17:00", "UnOccupied")],
        "wednesday": [("3:00", "Occupied"), ("17:00", "UnOccupied")],
        "thursday": [("4:00", "Occupied"), ("17:00", "UnOccupied")],
        "friday": [("5:00", "Occupied"), ("17:00", "UnOccupied")],
        "saturday": [("6:00", "Occupied"), ("17:00", "UnOccupied")],
        "sunday": [("7:00", "Occupied"), ("17:00", "UnOccupied")],
    },
}
schedule_example_binary = {
    "states": {"inactive": 0, "active": 1},
    "week": {
        "monday": [("1:00", "active"), ("16:00", "inactive")],
        "tuesday": [("2:00", "active"), ("16:00", "inactive")],
```

(continues on next page)

(continued from previous page)

```
        "wednesday": [("3:00", "active"), ("16:00", "inactive")],
        "thursday": [("4:00", "active"), ("16:00", "inactive")],
        "friday": [("5:00", "active"), ("16:00", "inactive")],
        "saturday": [("6:00", "active"), ("16:00", "inactive")],
        "sunday": [("7:00", "active"), ("16:00", "inactive")],
    },
}
schedule_example_analog = {
    "states": "analog",
    "week": {
        "monday": [("1:00", 22), ("18:00", 19)],
        "tuesday": [("2:00", 22), ("18:00", 19)],
        "wednesday": [("3:00", 22), ("18:00", 19)],
        "thursday": [("4:00", 22), ("18:00", 19)],
        "friday": [("5:00", 22), ("18:00", 19)],
        "saturday": [("6:00", 22), ("18:00", 19)],
        "sunday": [("7:00", 22), ("18:00", 19)],
    },
}
```

Note

Those examples are all available by calling *bacnet.schedule_example_analog* or *bacnet.schedule_example_binary* or *bacnet.schedule_example_multistate*. This make a quick way to get access to a template.

2.11.2 Missing informations

Those templates, are simple models to be edited and should be used to write to schedules. But when you read weeklySchedule from a controller, you will notice that more information will be retrieved.

No worries, you can take what's coming from the controller, edit the week events and send it back. When writing to a weeklySchedule BAC0 will only use the **states** and **week** element of the dict.

Example of a weeklySchedule from a controller

```
{'object_references': [('multiStateValue', 59)],
'references_names': ['OCC-SCHEDULE'],
'states': {'Occupied': 1, 'UnOccupied': 2, 'Standby': 3, 'Not Set': 4},
'reliability': 'noFaultDetected',
'priority': 15,
'presentValue': 'UnOccupied (2)',
'week': {'monday': [('01:00', 'Occupied'), ('17:00', 'UnOccupied')],
        'tuesday': [('02:00', 'Occupied'), ('17:00', 'UnOccupied')],
        'wednesday': [('03:00', 'Occupied'), ('17:00', 'UnOccupied')],
        'thursday': [('04:00', 'Occupied'), ('17:00', 'UnOccupied')],
        'friday': [('05:00', 'Occupied'), ('17:00', 'UnOccupied')],
        'saturday': [('06:00', 'Occupied'), ('17:00', 'UnOccupied')],
        'sunday': [('07:00', 'Occupied'), ('17:00', 'UnOccupied')]}}
```

As you can see, more information is available and can be used.

2.11.3 How things work

The ScheduleObject itself doesn't give all the information about the details and to build this representation, multiple network reading will occur.

2.11.3.1 object_references

The ScheduleObject will provide a list of Object property references. Those are the points inside the controller connected to the schedule.

2.11.3.2 references_names

For clarity, the names of the point, in the same order than the object_references so it's easy to tell which point is controlled by this scedule

2.11.3.3 States

BAC0 will read the first object_references and retrieve the states from this point. This way, we'll know the meaning of the integer values inside the schedule itself. "Occupied" is clearer than "1".

When using an **analog** schedule. States are useless as the value will consists on a floating value. If using an analog schedule, *states = 'analog'*.

When using **binary** schedules, BAC0 will consider fixed states (standard binary terms) *['inactive': 0, 'active' : 1]*

2.11.3.4 reliability

This is the reliability property of the schedule object exposed here for information

2.11.3.5 priority

This is the **priorityForWriting** property of the schedule. This tells at what priority the schedule will write to a point linked to the schedule (see object_references). If you need to override the internal schedule, you will need to use a higher priority for your logic to work.

2.11.3.6 PresentValue

Lnowing the states, BAC0 will give both the value and the name of the state for the presentValue.

2.11.3.7 week

This is the core of the weeklySchedule. This is a dict containing all days of the week (from monday to sunday, the order is VERY important. Each day consists of a list of event presented as tuple containing a string representation of the time and the value

```
{'monday': [('00:00', 'UnOccupied'),('07:00', 'Occupied'), ('17:00', 'UnOccupied')],
'tuesday': [('07:00', 'Occupied'), ('17:00', 'UnOccupied')],
'wednesday': [('07:00', 'Occupied'), ('17:00', 'UnOccupied')],
'thursday': [('07:00', 'Occupied'), ('17:00', 'UnOccupied')],
```

(continues on next page)

(continued from previous page)

```
'friday': [('07:00', 'Occupied'), ('17:00', 'UnOccupied')],
'saturday': [],
'sunday': []}}
```

2.11.4 Writing to the weeklySchedule

When your schedule dict is created, simply send it to the controller schedule by providing the address and the instance number of the schedule on which you want to write

```
bacnet.write_weeklySchedule("2:5", 10001, schedule)
```

2.12 COV in BAC0

BACnet supports a change of value (COV) mechanism that allow to subscribe to a device point to get notified when the value of this point changes.

In BAC0, you can subscribe to a COV from a point directly

```
device['point'].subscribe_cov()
```

or from the network itself

```
bacnet.cov(address, objectID)
```

> **Note**
>
> objectID is a tuple created with the object type as a string and the instance. For example analog input 1 would be : *("analogInput", 1)*

2.12.1 Confirmed COV

If the device to which you want to subscribe a COV supports it, it is possible to use a *confirmed* COV. In this case, the device will wait for a confirmation that you received the notification. This is the default case for BAC0.

To disable this, just pass *confirmed=False* to the subscribe_cov function.

2.12.2 Lifetime

COV subscription can be restricted in time by using the *lifetime* argument. By default, this is set to None (unlimited).

2.13 Callback

It can be required to call a function when a COV notification is received. This is done by providing the function as a callback to the subscription

```
# The Notification will pass a variable named "elements" to the callback
# your function must include this argument

# elements is a dict containing all the information of the COV
def my_callback(elements):
    print("Present value is : {}".format(elements['properties']['presentValue']))
```

Note

Here you can find a typical COV notification and the content of elements. {'source': <RemoteStation 2:6>, 'object_changed': ('analogOutput', 2131), 'properties': {'presentValue': 45.250762939453125, 'statusFlags': [0, 0, 0, 0]}}

2.14 Saving your data

When doing tests, it can be useful to go back in time and see what happened before. BAC0 allows you to save your progress (historical data) to a file that you'll be able to re-open in your device later.

Use

```
controller.save()
```

and voila! Two files are created. One (an SQLite file) contains all the histories, and one binary file containing all the details and properties of the device so the details can be rebuilt when needed.

By default, the 'object name' of the device is used as the filename. But you can specify a name

```
controller.save(db='new_name')
```

2.14.1 Offline mode

As already explained, a device in BAC0, if not connected (or cannot be reached) will be created as an offline device. If a database exists for this device, it will automatically loaded and all the points and histories will be available just as if if you were actually connected to the network.

You can also force a connection to use an existing database if needed. Provide connect function with the desired database's name.:

```
controller.connect(db='db_name')
```

Please note: this feature is experimental.

2.14.2 Saving Data to Excel

Thought the use of the Python module xlwings [https://www.xlwings.org/], it's possible to export all the data of a controller into an Excel Workbook.

Example

```
controller.to_excel()
```

2.15 Database

By default, all data is saved on a SQLite instance where BAC0 run. In some circumstances, it could be required to send data to a more powerful database. For that reason, support for [InfluxDB](https://docs.influxdata.com/influxdb/v2.0/) have been added to BAC0. I'm trying to make that flexible to allow other databases to be use eventually, using the same db_params argument when creating the network object.

This is still a work in progress.

2.15.1 SQL

Technically, BAC0 sends everything to SQLite locally. It would be possible to make some configuration changes to connect to a SQL database as SQLite share mostly the same commands (This is not actually implemented). Even if another databse is configured, the local SQLite file will be used.

2.15.2 InfluxDB

Work is done using InfluxDB v2.0 OSS. My setup is a RaspberryPi 4 running Ubuntu Server 64-bit

InfluxDB is installed on the RPi using default options. BAC0 will point to a Bucket (ex. named BAC0) using a token created in the InfluxDB web interface (ex. http://ip_of_rpi:8086)

To create the dashbpard, I use [Grafana](https://grafana.com/oss/) which is also installed on the same RaspberryPi. (ex. http://ip_of_rpi:3000)

> **Note**
>
> The python client used works also for InfluxDB v1.8+. Connecting to this version is supported and you must pass a username and a password in db_params

2.15.2.1 Connection

For BAC0 to connect to the inlfuxDB server, it needs to know where to send the data. This information can be given by using a dict

```
_params = {"name": "InfluxDB",
           "url" : "http://ip_of_rpi",
           "port" : 8086,
           "token" : "token_created in influxDB web interface",
           "org" : "the organization you created",
           "bucket" : "BAC0"
```

(continues on next page)

(continued from previous page)

```
            # (V1.8) "user" : " ",
            # (v1.8) "password" : "",
            }
```

Then you pass this information when you instanciate *bacnet*

    bacnet = BAC0.lite(db_params=_params)

The information can also be provided as environment variables. In that case, you must still provide name and bucket

```
_params = {"name": "InfluxDB",
           "bucket" : "BAC0"
           }
```

To use environment variables, BAC0 will count on python-dotenv to load a .env file in the folder when BAC0 is used.

The .env file must contain

```
# InfluxDB Params Example .env file
INFLUXDB_V2_URL=http://192.168.1.10:8086
INFLUXDB_V2_ORG=my-org
INFLUXDB_V2_TOKEN=123456789abcdefg
# INFLUXDB_V2_TIMEOUT=
# INFLUXDB_V2_VERIFY_SSL=
# INFLUXDB_V2_SSL_CA_CERT=
# INFLUXDB_V2_CONNECTION_POOL_MAXSIZE=
# INFLUXDB_V2_AUTH_BASIC=
# INFLUXDB_V2_PROFILERS=
```

Note

The name parameters in db_params would be use if any other implementation is made for another product. For now, only InfluxDB is valid.

2.15.2.2 Write Options configuration

Other options can be provided in the db_parmas dict to fine tune the configuration of the write_api.

- batch_size (default = 25)
- flush_interval (default =10 000)
- jitter_interval (default = 2 000)
- retry_interval (default = 5 000)
- max_retries (default = 5)
- max_retry_delay (default = 30 000)
- exponential_base (default = 2)

Please refer to InfluxDB documentation for all the details regarding those parameters.

ex.

```
_params = {"name": "InfluxDB",
        "bucket" : "BAC0",
        "batch_size" : 25,
        "flush_interval" : 10000,
        "jitter_interval" : 2000,
        "retry_interval" : 5000,
        "max_retries" : 5,
        "max_retry_delay" : 30000,
        "exponential_base" : 2,
       }
```

2.15.2.3 Timestamp

Now all timestamps in BAC0 will be timezone aware. As long as you are using in-memory data, the actual timezone will be used. I didn't want to mess with the timestamp for day to day work requiring only quick histories and minor tests. But all timestamps that will be sent to InfluxDB will be converted to UTC. This is a requirement and makes things work well with Grafana.

2.15.2.4 API

BAC0 will use the Python package named influxdb-client, which must be pip installed.

pip install 'influxdb-client'

Refer to [documentation](https://github.com/influxdata/influxdb-client-python) for details.

In my actual tests, I haven't work with ciso8601, RxPy neither.

The API will accumulate write requests and write them in batch that are configurable. The actual implementation use 25 as the batch parameters. This is subject to change.

2.15.2.5 Write all

I have included a function that write all histories to InfluxDB. This function takes all the Pandas Series and turn them into a DataFrame which is then sent to InfluxDB.

I'm not sure if it's really useful as the polling takes care of sending the data constantly.

2.15.2.6 Write to the database

Each call to *_trend* (which add a record in memory) will call a write request to the API if the database is defined.

2.15.2.7 ID of the record

The ID of the record will be

```
Device_{device_id}/{object}
```

For example

```
Device_5004/analogInput:1
```

This choice was made to make sure all records ID were unique as using name could lead to errors. As name, device name, etc are provided as tags, I suggest using them in the Flux requests.

2.15.2.8 Tags and fields

InfluxDB allows the usage of tags and multiple fields for values. This allows making requests based on tags when creating dashboard. I chose to add some information in the form of tags when writing to the database :

- object_name
- description
- units_state (units of measure or state text for multiState and Binary)
- object instance (ex. analogInput:1)
- device_name (the name of the controller)
- device_id (the device instance)

2.15.2.9 value

Two value fields are included. A value field and a string_value field. This way, when working with binary or multistate, it's possible to use aggregation functions using the numerical value (standard value), but it is also possible to make database request on the string_value field and get a more readable result (ex. Occupied instead of 0)

2.16 Local Objects

Until now, we've looked into the capability of BAC0 to act as a "client" device. This is not a very good term in the context of BACnet but at least it is easy to understand. But BAC0 can also act as a BACnet device. It can be found in a network and will present its properties to every other BACnet devices.

What if you want to create a BACnet device with objects with BAC0 ?

You will need to provide local objects that will be added to the application part of the BAC0 instance. This part is called *this_application*.

2.16.1 What are BACnet objects

BACnet objects are very diverse. You probably know the main ones like AnalogValue or BinaryInput or AnalogOutput, etc. But there are more. Depending on your needs, there will be an object that fit what you try to define in your application. The complet definition of all BACnet objects fall outside the scope of this document. Please refer to the BACnet standard if you want to know more about them. For the sake of understanding, I'll cover a few of them.

The definition of the BACnet obejcts is provided by *bacpypes.object*. You will need to import the different classes of objects you need when you will create the objects.

An object, like you probably know now, owes properties. Those properties can be read-only, writable, mandatory or optional. This is defined in the standard. Typically, the actual value of an object is given by a property name *presentValue*. Another property called *relinquishDefault* would hold the value the object should give as a presentValue when all other priorityArray are null. PriorityArray is also a property of an object. When you create an object, you must know which properties you must add to the object and how you will interact with thoses properties and how they will interact with one another.

This makes a lot to know when you first want to create one object.

2.16.2 A place to start

The enormous complexity of BACnet objects led me to think of a way to generate objects with a good basis. Just enough properties depending on the commandability of the object (do you need other devices to write to those objects ?). This decision hAVe an impact on the chosen properties of a BACnet object.

For any commandable object, it would be reasonable to provide a priorityArray and a relinquishDefault. Those propoerties make no sense for a non-commandable object.

For any analog object, an engineering unit should be provided.

Those basic properties, depending on the type of objects, the BAC0's user should not hAVe to think about them. They should just be part of the object.

2.16.3 Working in the factory

BAC0 will allow the creation of BACnet objects with a special class named ObjectFactory. This class will take as argument the objectType, instance, name, description, properties, etc and create the object. This object will then be added to a class variable (a dict) that will be used later to populate the application will all the created objects.

The factory will take as an argument *is_commandable* (boolean) and will modify the base type of object to make it commandable if required. This part is pretty complex as a subclass with a Commandable mixin must be created fot each objectType. ObjectFactory uses a special decorator that will recreate a new subclass with eveything that is needed to make the point commandable.

Another decorator will also allow the addition of custom properties (that would not be provided by default) if it's required.

Another decorator will allow the addition of "features" to the objects. Thos will need to be defined but we can think about event generation, alarms, MinOfOff behAViour, etc.

The user will not hAVe to think about the implementation of the decorators as everything is handled by the ObjectFactory. But that said, nothing prevent you to create your own implementation of a factory using those decorators.

2.16.4 An example

A good way to understand how things work is by giving an example. This code is part of the tests folder and will give you a good idea of the way objects can be defined inside a BAC0's instance

```
def build():
    bacnet = BAC0.lite(deviceId=3056235)

    new_obj = ObjectFactory(
        AnalogValueObject,
        0,
        "AV0",
        properties={"units": "degreesCelsius"},
        presentValue=1,
        description="Analog Value 0",
    )
    ObjectFactory(
        AnalogValueObject,
        1,
        "AV1",
        properties={"units": "degreesCelsius"},
```

(continues on next page)

(continued from previous page)

```
        presentValue=12,
        description="Analog Value 1",
        is_commandable=True,
    )
    ObjectFactory(
        CharacterStringValueObject,
        0,
        "cs0",
        presentValue="Default value",
        description="String Value 0",
    )
    ObjectFactory(
        CharacterStringValueObject,
        1,
        "cs1",
        presentValue="Default value",
        description="Writable String Value 1",
        is_commandable=True,
    )

    new_obj.add_objects_to_application(bacnet.this_application)
    return bacnet
```

2.17 Models

So it's possible to create objects but even using the object factory, things are quite complex and you need to cover a lot of edge cases. What if you want to create a lot of similar objects. What if you need to be sure each one of them will hAVe the basic properties you need.

To go one step further, BAC0 offers models that can be used to simplify (at least to try to simplify) the creation of local objects.

Models are an opiniated version of BACnet objects that can be used to create the objects you need in your device. There are still some features that are not implemented but a lot of features hAVe been covered by those models.

Models use the ObjectFactory but with a supplemental layer of abstraction to provide basic options to the objects.

For example, "analog" objects hAVe common properties. But the objectType will be different if you want an analogInput or an analogValue. By default, AnalogOutput will be commandable, but not the analogInput (not in BAC0 at least as it doesn't support behAViour that allows to write to the presentValue when the out_of_service property is True). Instead of letting the user thinking about all those details, you can simply create an *analogInput* and BAC0 will take care of the details.

Actually, BAC0 implements those models :

> analog_input, analog_output, analog_value, binary_input, binary_output, binary_value, multistate_input, multistate_output, multistate_value, date_value, datetime_value, temperature_input, temperature_value, humidity_input, humidity_value, character_string,

Again, the best way to understand how things work, is by looking at code sample :

> # code here

2.18 State Text

One important feature for multiState values is the state text property. This define a text to shown in lieu of an integer. This adds a lot of clarity to those objects. A device can tell a valve is “Open/Close”, a fan is “Off/On”, a schedule is “Occupied/Unoccupied/Stanby/NotSet”. It brings a lot of value.

To define state text, you must use the special function with a list of states then you pass this variable to the properties dict :

states = make_state_text([“Normal”, “Alarm”, “Super Emergency”]) _new_object = multistate_value(

description=”An Alarm Value”, properties={“stateText”: states}, name=”BIG-ALARM”, is_commandable=True,

)

2.19 Engineering units

Valid Engineering untis to be used are :

ampereSeconds ampereSquareHours ampereSquareMeters amperes amperesPerMeter amperesPerSquareMeter bars becquerels btus btusPerHour btusPerPound btusPerPoundDryAir candelas candelasPerSquareMeter centimeters centimetersOfMercury centimetersOfWater cubicFeet cubicFeetPerDay cubicFeetPerHour cubicFeetPerMinute cubicFeetPerSecond cubicMeters cubicMetersPerDay cubicMetersPerHour cubicMetersPerMinute cubicMetersPerSecond currency1 currency10 currency2 currency3 currency4 currency5 currency6 currency7 currency8 currency9 cyclesPerHour cyclesPerMinute days decibels decibelsA decibelsMillivolt decibelsVolt degreeDaysCelsius degreeDaysFahrenheit degreesAngular degreesCelsius degreesCelsiusPerHour degreesCelsiusPerMinute degreesFahrenheit degreesFahrenheitPerHour degreesFahrenheitPerMinute degreesKelvin degreesKelvinPerHour degreesKelvinPerMinute degreesPhase deltaDegreesFahrenheit deltaDegreesKelvin farads feet feetPerMinute feetPerSecond footCandles grams gramsOfWaterPerKilogramDryAir gramsPerCubicCentimeter gramsPerCubicMeter gramsPerGram gramsPerKilogram gramsPerLiter gramsPerMilliliter gramsPerMinute gramsPerSecond gramsPerSquareMeter gray hectopascals henrys hertz horsepower hours hundredthsSeconds imperialGallons imperialGallonsPerMinute inches inchesOfMercury inchesOfWater jouleSeconds joules joulesPerCubicMeter joulesPerDegreeKelvin joulesPerHours joulesPerKilogramDegreeKelvin joulesPerKilogramDryAir kiloBtus kiloBtusPerHour kilobecquerels kilograms kilogramsPerCubicMeter kilogramsPerHour kilogramsPerKilogram kilogramsPerMinute kilogramsPerSecond kilohertz kilohms kilojoules kilojoulesPerDegreeKelvin kilojoulesPerKilogram kilojoulesPerKilogramDryAir kilometers kilometersPerHour kilopascals kilovoltAmpereHours kilovoltAmpereHoursReactive kilovoltAmperes kilovoltAmperesReactive kilovolts kilowattHours kilowattHoursPerSquareFoot kilowattHoursPerSquareMeter kilowattHoursReactive kilowatts liters litersPerHour litersPerMinute litersPerSecond lumens luxes megaBtus megabecquerels megahertz megajoules megajoulesPerDegreeKelvin megajoulesPerKilogramDryAir megajoulesPerSquareFoot megajoulesPerSquareMeter megAVoltAmpereHours megAVoltAmpereHoursReactive megAVoltAmperes megAVoltAmperesReactive megAVolts megawattHours megawattHoursReactive megawatts megohms meters metersPerHour metersPerMinute metersPerSecond metersPerSecondPerSecond microSiemens microgramsPerCubicMeter microgramsPerLiter microgray micrometers microsieverts microsievertsPerHour milesPerHour milliamperes millibars milligrams milligramsPerCubicMeter milligramsPerGram milligramsPerKilogram milligramsPerLiter milligray milliliters millilitersPerSecond millimeters millimetersOfMercury millimetersOfWater millimetersPerMinute millimetersPerSecond milliohms milliseconds millisiemens millisieverts millivolts milliwatts minutes minutesPerDegreeKelvin months nanogramsPerCubicMeter nephelometricTurbidityUnit newton newtonMeters newtonSeconds newtonsPerMeter noUnits ohmMeterPerSquareMeter ohmMeters ohms pH partsPerBillion partsPerMillion pascalSeconds pascals perHour perMille perMinute perSecond percent percentObscurationPerFoot percentObscurationPerMeter percentPerSecond percentRelative-

Humidity poundsForcePerSquareInch poundsMass poundsMassPerHour poundsMassPerMinute poundsMassPerSecond powerFactor psiPerDegreeFahrenheit radians radiansPerSecond revolutionsPerMinute seconds siemens siemensPerMeter sieverts squareCentimeters squareFeet squareInches squareMeters squareMetersPerNewton teslas therms tonHours tons tonsPerHour tonsRefrigeration usGallons usGallonsPerHour usGallonsPerMinute voltAmpereHours voltAmpereHoursReactive voltAmperes voltAmperesReactive volts voltsPerDegreeKelvin voltsPerMeter voltsSquareHours wattHours wattHoursPerCubicMeter wattHoursReactive watts wattsPerMeterPerDegreeKelvin wattsPerSquareFoot wattsPerSquareMeter wattsPerSquareMeterDegreeKelvin webers weeks years

2.20 Web Interface

2.20.1 Flask App

BAC0 when used in "Complete" mode will start a Web app that can be reached with a browser. The app will present the bokeh server feature for live trending.

More documentation will come in the future as this feature is under development.

2.21 Demo in a Jupyter Notebook

When installed, module can be used to script communication with bacnet device. Jupyter Notebooks are an excellent way to test it. Here is an example.

2.22 Testing and simulating with BAC0

BAC0 is a powerful BAS test tool. With it you can easily build tests scripts, and by using its **assert** syntax, you can make your DDC code stronger.

2.22.1 Using Assert and other commands

Let's say your BAC controller **sequence of operation** is really simple. Something like this:

```
System stopped:
    When system is stopped, fan must be off,
    dampers must be closed, heater cannot operate.

System started:
    When system starts, fan command will be on.
    Dampers will open to minimum position.
    If fan status turns on, heating sequence will start.
```

And so on…

2.22.2 How would I test that ?

Assuming:

- Controller is defined and its variable name is mycontroller
- fan command = SF-C
- Fan Status = SF-S
- Dampers command = MAD-O
- Heater = RH-O
- Occupancy command = OCC-SCHEDULE

System Stopped Test Code:

```
mycontroller['OCC-SCHEDULE'] = Unoccupied
time.sleep(10)
assert mycontroller['SF-C'] == False
assert mycontroller['MAD-O'] == 0
assert mycontroller['RH-O'] == 0

# Simulate fan status as SF-C is Off
mycontroller['SF-S'] = 'Off'
```

Sytstem Started Test Code:

```
mycontroller['OCC-SCHEDULE'] = 'Occupied'
time.sleep(10)
assert mycontroller['SF-C'] == 'On'
# Give status
mycontroller['SF-S'] = 'On'
time.sleep(15)
assert mycontroller['MAD-O'] == mycontroller['MADMIN-POS']
```

And so on…

You can define any test you want. As complex as you want. You will use more precise conditions instead of a simple time.sleep() function - most likely you will read a point value that tells you when the actual mode is active.

You can then add tests for the various temperature ranges; and build functions to simulate discharge air temperature depending on the heating or cooling stages… it's all up to you!

2.23 Using tasks to automate simulation

2.23.1 Polling

Let's say you want to poll a point every 5 seconds to see how the point reacted.:

```
mycontroller['point_name'].poll(delay=5)
```

Note: by default, polling is enabled on all points at a 10 second frequency. But you could define a controller without polling and do specific point polling.

mycontroller = BAC0.device('2:5',5,bacnet,poll=0) mycontroller['point_name'].poll(delay=5)

2.23.2 Match

Let's say you want to automatically match the status of a point with it's command to find times when it is reacting to conditions other than what you expected.:

```
mycontroller['status'].match(mycontroller['command'])
```

2.23.3 Custom function

You could also define a complex function, and send that to the controller. This way, you'll be able to continue using all synchronous functions of Jupyter Notebook for example. (technically, a large function will block any inputs until it's finished)

> **Note**
>
> THIS IS A WORK IN PROGRESS

Example

```
import time

def test_Vernier():
    for each in range(0,101):
        controller['Vernier Sim'] = each
        print('Sending : %2f' % each)
        time.sleep(30)

controller.do(test_Vernier)
```

This function updates the variable named "Vernier Sim" each 30 seconds; incrementing by 1 percent. This will take a really long time to finish. So instead, use the "do" method, and the function will be run is a separate thread so you are free to continue working on the device, while the function commands the controller's point.

2.24 Using Pytest

Pytest [https://docs.pytest.org/en/latest/] is a "a mature full-featured Python testing tool". It allows the creation of test files that can be called by a command line script, and run automatically while you work on something else.

For more details, please refer Pytest's documentation.

2.24.1 Some basic stuff before we begin

Pytest is a very simple testing tool. While, the default unit test tool for python is **unittest** (which is more formal and has more features); unittest can easily become too much for the needs of testing DDC controllers.

Pytest uses only simple the *assert* command, and locally defined functions. It also allows the usage of "fixtures" which are little snippets of code that prepare things prior to the test (setUp), then finalize things when the test is over (tearDown).

The following example uses fixtures to establish the BACnet connection prior to the test, and then saves the controller histories and closes the connection after the tests are done.

2.24.1.1 Example

Code

```
import BAC0
import time
import pytest

# Make a fixture to handle connection and close it when it's over
@pytest.fixture(scope='module')
def bacnet_network(request):
    print("Let's go !")
    bacnet = BAC0.connect()
    controller = BAC0.device('2:5', 5, bacnet)

    def terminate():
        controller.save()
        bacnet.disconnect()
        print('It's over')
    request.addfinalizer(terminate)
    return controller

def test_input1_is_greater_than_zero(bacnet_network):
    assert controller['nvoAI1'] > 0

def test_input2_equals_fifty(bacnet_network):
    assert controller['nvoAI2'] > 0

def test_stop_fan_and_check_status_is_off(bacnet_network):
    controller['SF-C'] = False
    time.sleep(2)
    assert controller['SF-S'] == False

def test_start_fan_and_check_status_is_on(controller):
    controller['SF-C'] = True
    time.sleep(2)
    assert controller['SF-S'] == True
```

2.24.1.1.1 Success result

If you name the file: test_mytest.py, you can just run

```
py.test -v -s
```

Pytest will look for the test files, find them and run them. Or you can define the exact file you want to run

```
py.test mytestfile.py -v -s
```

Here's what it looks like

```
============================= test session starts ==============================
platform win32 -- Python 3.4.4, pytest-2.8.5, py-1.4.31, pluggy-0.3.1 -- C:\User
```

(continues on next page)

(continued from previous page)

```
s\ctremblay.SERVISYS\AppData\Local\Continuum\Anaconda3\python.exe
cachedir: .cache
rootdir: c:\0Programmes\Github\BAC0, inifile:
plugins: bdd-2.16.1, cov-2.2.1, pep8-1.0.6
collected 2 items

pytest_example.py::test_input1_is_greater_than_zero Let's go !
Using ip : 192.168.210.95
Starting app...
App started
Starting Bokeh Serve
Click here to open Live Trending Web Page
http://localhost:5006/?bokeh-session-id=um2kEfnM97alVOr3GRu5xt07hvQItkruMVUUDpsh
S8Ha
Changing device state to <class 'BAC0.core.devices.Device.DeviceDisconnected'>
Changing device state to <class 'BAC0.core.devices.Device.RPMDeviceConnected'>
Found FX14 0005... building points list
Failed running bokeh.bat serve
Bokeh server already running
Ready!
Polling started, every values read each 10 seconds
PASSED
pytest_example.py::test_input2_equals_fifty PASSEDFile exists, appending data...

FX14 0005 saved to disk
Stopping app
App stopped
It's over


========================== 2 passed in 27.94 seconds ==========================
```

2.24.1.1.2 Failure result

Here's what a test failure looks like:

```
============================= test session starts ==============================
platform win32 -- Python 3.4.4, pytest-2.8.5, py-1.4.31, pluggy-0.3.1 -- C:\User
s\ctremblay.SERVISYS\AppData\Local\Continuum\Anaconda3\python.exe
cachedir: .cache
rootdir: c:\0Programmes\Github\BAC0, inifile:
plugins: bdd-2.16.1, cov-2.2.1, pep8-1.0.6
collected 2 items

pytest_example.py::test_input1_is_greater_than_zero Let's go !
Using ip : 192.168.210.95
Starting app...
App started
Starting Bokeh Serve
Click here to open Live Trending Web Page
http://localhost:5006/?bokeh-session-id=TKgDiRoCkut2iobSFRlWGA2nhJlPCtXU3ZTWL3cC
```

(continues on next page)

(continued from previous page)

```
nxRI
Changing device state to <class 'BAC0.core.devices.Device.DeviceDisconnected'>
Changing device state to <class 'BAC0.core.devices.Device.RPMDeviceConnected'>
Found FX14 0005... building points list
Failed running bokeh.bat serve
Bokeh server already running
Ready!
Polling started, every values read each 10 seconds
PASSED
pytest_example.py::test_input2_equals_fifty FAILEDFile exists, appending data...

FX14 0005 saved to disk
Stopping app
App stopped
It's over


================================== FAILURES ===================================
_____________________________ test_input2_equals_fifty ______________________________

controller = FX14 0005 / Connected

    def test_input2_equals_fifty(controller):
>       assert controller['nvoAI2'] > 1000
E       assert nvoAI2 : 20.58 degreesCelsius > 1000

pytest_example.py:30: AssertionError
====================== 1 failed, 1 passed in 30.71 seconds ======================
```

Note: I modified the test to generate an failure - nvoAI2 cannot exceed 1000.

2.24.2 Conclusion

Using Pytest is a really good way to generate test files that can be reused and modified depending on different use cases. It's a good way to run multiple tests at once. It provides concise reports of every failure and tells you when your tests succeed.

2.25 Logging and debugging

All interactions with the user in the console is made using logging and an handler. Depending on the user desire, the level can be adjusted to limit or extend the verbosity of the app.

It is not recommended to set the stdout to logging.DEBUG level as it may fill the shell with messages and make it very hard to enter commands. Typically, 'debug' is sent to the file (see below).

By default, stderr is set to logging.CRITICAL and is not used; stdout is set to logging.INFO; file is set to logging.WARNING. The goal behind to not fill the file if it is not explicitly wanted.

2.25.1 Level

You can change the logging level using

```
import BAC0
BAC0.log_level(level)
# level being 'debug, info, warning, error'
# or
BAC0.log_leve(log_file=logging.DEBUG, stdout=logging.INFO, stderr=logging.CRITICAL)
```

2.25.2 File

A log file will be created under your user folder (~) / .BAC0 It will contain warnings by default until you change the level.

Extract from the log file (with INFO level entries)

```
2018-04-08 21:42:45,387 - INFO    | Starting app...
2018-04-08 21:42:45,390 - INFO    | BAC0 started
2018-04-08 21:47:21,766 - INFO    | Changing device state to <class 'BAC0.core.devices.
↪Device.DeviceDisconnected'>
2018-04-08 21:47:21,767 - INFO    |
2018-04-08 21:47:21,767 - INFO    | ###################################
2018-04-08 21:47:21,767 - INFO    | # Read property
2018-04-08 21:47:21,768 - INFO    | ###################################
2018-04-08 21:47:22,408 - INFO    | value                datatype
2018-04-08 21:47:22,409 - INFO    | 'FX14 0005'          <class 'bacpypes.primitivedata.
↪CharacterString'>
2018-04-08 21:47:22,409 - INFO    |
2018-04-08 21:47:22,409 - INFO    | ###################################
2018-04-08 21:47:22,409 - INFO    | # Read property
2018-04-08 21:47:22,409 - INFO    | ###################################
2018-04-08 21:47:23,538 - INFO    | value                datatype
2018-04-08 21:47:23,538 - INFO    | 'segmentedTransmit'  <class 'bacpypes.basetypes.
↪Segmentation'>
2018-04-08 21:47:23,538 - INFO    | Changing device state to <class 'BAC0.core.devices.
↪Device.RPMDeviceConnected'>
2018-04-08 21:47:29,510 - INFO    | ###################################
2018-04-08 21:47:29,510 - INFO    | # Read Multiple
2018-04-08 21:47:29,511 - INFO    | ###################################
2018-04-08 21:47:30,744 - INFO    |
2018-04-08 21:47:30,744 - INFO    |␣
↪==========================================================================================
2018-04-08 21:47:30,744 - INFO    | 'analogValue' : 15
2018-04-08 21:47:30,744 - INFO    |␣
↪==========================================================================================
2018-04-08 21:47:30,745 - INFO    | propertyIdentifier   propertyArrayIndex   value     ␣
↪                    datatype
2018-04-08 21:47:30,745 - INFO    | ------------------------------------------------
↪------------------------------------------------------------
2018-04-08 21:47:30,745 - INFO    | 'objectName'         None
↪'nciPIDTPRdCTI'                <class 'bacpypes.primitivedata.CharacterString'>
2018-04-08 21:47:30,745 - INFO    | 'presentValue'       None                 800.0     ␣
```

(continues on next page)

(continued from previous page)

```
↪                    <class 'bacpypes.primitivedata.Real'>
2018-04-08 21:47:30,745 - INFO    | 'units'              None                 'seconds' ␣
↪                    <class 'bacpypes.basetypes.EngineeringUnits'>
2018-04-08 21:47:30,746 - INFO    | 'description'        None
↪'nciPIDTPRdCTI'                <class 'bacpypes.primitivedata.CharacterString'>
2018-04-10 23:18:26,184 - DEBUG   | BAC0.core.app.ScriptApplication |␣
↪ForeignDeviceApplication | ('do_IAmRequest %r', <bacpypes.apdu.IAmRequest(0) instance␣
↪at 0x9064c88>)
```

CHAPTER

THREE

DEVELOPER DOCUMENTATION

3.1 BAC0

3.1.1 BAC0 package

3.1.1.1 Subpackages

3.1.1.1.1 BAC0.core package

Subpackages

BAC0.core.app package

Submodules

BAC0.core.app.ScriptApplication module

Module contents

BAC0.core.devices package

Subpackages

BAC0.core.devices.local package

Submodules

BAC0.core.devices.local.decorator module

BAC0.core.devices.local.object module

Module contents

BAC0.core.devices.mixins package

Submodules

BAC0.core.devices.mixins.CommandableMixin module

BAC0.core.devices.mixins.read_mixin module

Module contents

Submodules

BAC0.core.devices.Device module

BAC0.core.devices.Points module

BAC0.core.devices.Trends module

BAC0.core.devices.create_objects module

Module contents

BAC0.core.functions package

Submodules

BAC0.core.functions.DeviceCommunicationControl module

BAC0.core.functions.Discover module

BAC0.core.functions.GetIPAddr module

BAC0.core.functions.Reinitialize module

BAC0.core.functions.TimeSync module

Module contents

BAC0.core.io package

Submodules

BAC0.core.io.IOExceptions module

IOExceptions.py - BAC0 application level exceptions

exception BAC0.core.io.IOExceptions.**APDUError**

Bases: Exception

exception BAC0.core.io.IOExceptions.**ApplicationNotStarted**

Bases: Exception

Application not started, no communication available.

exception BAC0.core.io.IOExceptions.**BadDeviceDefinition**

Bases: Exception

exception BAC0.core.io.IOExceptions.**BokehServerCantStart**

Bases: Exception

Raised if Bokeh Server can't be started automatically

exception BAC0.core.io.IOExceptions.**BufferOverflow**

Bases: Exception

Buffer capacity of device exceeded.

exception BAC0.core.io.IOExceptions.**DataError**

Bases: Exception

exception BAC0.core.io.IOExceptions.**DeviceNotConnected**

Bases: Exception

exception BAC0.core.io.IOExceptions.**InitializationError**

Bases: Exception

exception BAC0.core.io.IOExceptions.**NetworkInterfaceException**

Bases: Exception

This exception covers different network related exc eption (like finding IP or subnet mask...)

exception BAC0.core.io.IOExceptions.**NoResponseFromController**

Bases: Exception

This exception is used when trying to read or write and there is not answer.

exception BAC0.core.io.IOExceptions.**NotReadyError**

Bases: Exception

exception BAC0.core.io.IOExceptions.**NumerousPingFailures**

Bases: Exception

exception BAC0.core.io.IOExceptions.**OutOfServiceNotSet**

Bases: Exception

This exception is used when trying to simulate a point and the out of service property is false.

exception BAC0.core.io.IOExceptions.**OutOfServiceSet**

Bases: Exception

This exception is used when trying to set the out of service property to false to release the simulation...and it doesn't work.

exception BAC0.core.io.IOExceptions.**ReadPropertyException**

Bases: ValueError

This exception is used when trying to read a property.

exception BAC0.core.io.IOExceptions.ReadPropertyMultipleException
Bases: ValueError

This exception is used when trying to read multiple properties.

exception BAC0.core.io.IOExceptions.ReadRangeException
Bases: ValueError

This exception is used when trying to read a property.

exception BAC0.core.io.IOExceptions.RemovedPointException
Bases: Exception

When defining a device from DB it may not be identical to the actual device.

exception BAC0.core.io.IOExceptions.SegmentationNotSupported
Bases: Exception

exception BAC0.core.io.IOExceptions.Timeout
Bases: Exception

exception BAC0.core.io.IOExceptions.UnknownObjectError
Bases: Exception

exception BAC0.core.io.IOExceptions.UnknownPropertyError
Bases: Exception

exception BAC0.core.io.IOExceptions.UnrecognizedService
Bases: Exception

This exception is used when trying to read or write and there is not answer.

exception BAC0.core.io.IOExceptions.WriteAccessDenied
Bases: Exception

This exception is used when trying to write and controller refuse it.

exception BAC0.core.io.IOExceptions.WritePropertyCastError
Bases: Exception

This exception is used when trying to write to a property and a cast error occurs.

exception BAC0.core.io.IOExceptions.WritePropertyException
Bases: Exception

This exception is used when trying to write a property.

exception BAC0.core.io.IOExceptions.WrongParameter
Bases: Exception

BAC0.core.io.Read module

BAC0.core.io.Simulate module

BAC0.core.io.Write module

Module contents

Module contents

3.1.1.2 Submodules

3.1.1.3 BAC0.infos module

infos.py - BAC0 Package MetaData

3.1.1.4 Module contents

CHAPTER

FOUR

INDEX AND SEARCH TOOL

- genindex
- modindex
- search

PYTHON MODULE INDEX

b

INDEX

W