domogik-plugin-nutserve

Release 0.1

November 10, 2015

Contents

CHAPTER 1

Plugin documentation

1.1 Last change

New instructions to apply changes from immediatly previous version.

- **0.1a0** [(28-05-2014) First published version]
  - Target :
  - **Change log :**
    - Update doc
- Previous change

1.2 Purpose

This Domogik plugin monitor UPS (Uninterruptible Power Supplies) through communicat with NUT project server.

Create a socket connection with NUT to get UPS informations and format them to xPL format according to xPL_project specifications.

You must configure as your needs the NUT library and give parameters connection at plugin.

UPS Status, on mains (on line), on battery, connected, lost connection,.... are send to domogik device.

Sensors values for input, output, battery voltage and battery charge are also send to domogik device.

1.3 Plugin configuration

1.3.1 Configuration

In Domogik administration section, go to client plugin-nutserve details page.

Key	Default value	Description
startup-plugin	false	Automatically start plugin at Domogik startup
Ip nut server	localhost	Set host NUT server.
Port nut server	3493	Set NUT server port number.
Login		Set NUT user login.
Password		Set UT user password.

1.3.2 Creating devices for UPS Client

In clients page of admin UI, go to **plugin-nutserve-<your_host_domogik>**, select tab **"Devices"**, **"New"** to create your devices.

Chose one way creation by product or instance type.

Instance-type : ups.device

Key	Example	Description
Device	My_UPS	The display name for this device.
Description	What you want	A short descriptionn for this device.
Global device	My_UPS	Same name of UPS name in ups.conf file.
Global timer_poll	3	**Timer (seconds) for poll UPS status** • Default "0": get NUT Timer, desactivat if not find. • 31536000 for desactivat polling.

CHAPTER 2

Development informations

2.1 How it UPS status found ?

The plugin just use the NUT server functions.

2.2 xPL messages

xPL messages are according to xPL Project UPS specification

2.2.1 xpl-stat

n/a

2.2.2 xpl-trig

The ups.basic message is used

```
xpl-trig
{
...
}
ups.basic
{
device=<ups name declared on domogik device created>
status=<one of these values : mains, battery, unknown>
event=<last event : onmains, onbattery, battlow,......>
}
```

The sensor.basic message is used for additionnal data

```
xpl-trig
{
...
}
sensor.basic
{
```

```
device=<ups name declared on domogik device created>
input-voltage=<value>
output-voltage=<value>
battery-voltage=<value>
battery-charge==<value>
}
```

2.2.3 xpl-cmnd

n/a

2.3 NUT driver compatibility

Plugin need driver device class for working with specific UPS.

2.3.1 Existing drivers device.

- **Basic:** Must work with any NUT driver but handle only status and events `onmains`, `onbattery`, `batterylow`, `comm_ok` and `comm_lost` If NUT server return `battery.charge` driver can handle it.
- **Blazer_USB:** Handle all basic and if available by NUT server:

  `battery.charge` calculat, input-output voltage status , input frequence status, output current status and temperature status

2.3.2 Add developement NUT driver compatibility

Plugin offer simple way to add new driver device.

File `lib/nutdevices.py` contains a `DeciveBase` python class who handle basic methodes. You create a new class inherit this base class and overwriting some methodes to get NUT driver compatibility. You just have to add new driver device class reference in `createDevice` methode.

```
def createDevice(data):
    """ Create a device depending of 'driver.name' given by data dict.
        - Developer : add your python class derived from DeviceBase class."""
    if data.has_key('driver.name') :
        if data['driver.name'] == 'blazer_usb' : return Blazer_USB(data)
          # Just add next line
        elif data['driver.name'] == 'My_new_driver' : return My_New_Device(data)
        else : return DeviceBase(data)
    else : return DeviceBase(data)
```

Have a look to Network UPS Tools and get details informations on Driver section of User manual pages

Getting UPS variables available

- **Use clients commands upsc of Network UPS Tools to get all variables available :** Example :

```
$ /bin/upsc MyUPS@localhost
battery.voltage.nominal: 12.0
battery.voltage: 13.60
beeper.status: enabled
device.type: ups
driver.version.internal: 0.04
driver.name: blazer_usb
driver.version: 2.6.3
driver.parameter.pollinterval: 2
driver.parameter.port: /dev/upsZ3
driver.parameter.productid: 5161
driver.parameter.vendorid: 0665
input.voltage.nominal: 230
input.voltage.fault: 240.2
input.voltage: 240.2
input.current.nominal: 2.0
input.frequency.nominal: 50
input.frequency: 49.9
output.voltage: 240.2
ups.type: offline
ups.vendorid: 0665
ups.delay.shutdown: 30
ups.productid: 5161
ups.delay.start: 180
ups.status: OL
```

- **Or using plugin tools `lib/nutdevices.py` :** Simple set your own parameters on __main__ section of `lib/nutdevices.py` file.

```
if __name__ == "__main__":
      # Set  upsaddr,  upsport and upsname depending on your config
    upsaddr = <UPS server IP >
    upsport = <UPS server port (generaly= 3493)>
    upsname = <UPS name>
```

Run python file `lib/nutdevices.py` and you get appropriate informations.

```
$ python nutsockclient.py
 +++ Internal NUTSocketClient created +++
Client connected to Z3_SERVER

*** getNUTVersion
{'cmd': 'VER', 'data': '2.6.3', 'error': ''}

*** getNUTNetworkVersion
{'cmd': 'NETVER', 'error': 'NUT version to old (2.6.3), NETVER function not handle, Update N

*** getNUTHelp
Commands: HELP VER GET LIST SET INSTCMD LOGIN LOGOUT USERNAME PASSWORD STARTTLS

{}

*** getUPSList
{'cmd': 'LIST UPS', 'data': {'Z3_SERVER': 'Server Linux Internet'}, 'error': ''}

*** getUPSVars
{'cmd': 'LIST VAR',
    'data': {
        'input.voltage.nominal': '230', 'beeper.status': 'enabled', 'input.voltage.fault': '
```

```
                'driver.version.internal': '0.04', 'input.voltage': '238.4', 'ups.type': 'offline',
                'ups.vendorid': '0665', 'driver.name': 'blazer_usb', 'ups.delay.shutdown': '30', 'ou
                'ups.productid': '5161', 'ups.delay.start': '180', 'driver.version': '2.6.3', 'input
                'battery.voltage.nominal': '12.0', 'driver.parameter.pollinterval': '2', 'driver.par
                'battery.voltage': '13.60', 'driver.parameter.productid': '5161', 'ups.status': 'OL'
                'driver.parameter.vendorid': '0665'
                },
         'ups': 'Z3_SERVER', 'error': ''}

        *** getUPSRWVars
        {'cmd': 'LIST RW', 'data': {}, 'ups': 'Z3_SERVER', 'error': ''}

        *** getUPSCommands
        {'cmd': 'LIST CMD',
            'data': [
                'beeper.toggle', 'load.off', 'load.on', 'shutdown.return', 'shutdown.stayoff', 'shut
                'test.battery.start', 'test.battery.start.deep', 'test.battery.start.quick', 'test.b
                ],
         'ups': 'Z3_SERVER', 'error': ''}

        *** getUPSListClients
        {'cmd': 'LIST CLIENT', 'error': 'NUT version to old (2.6.3), LIST CLIENT function not handle

        *** getUPSVar
        {'var': 'ups.status', 'cmd': 'GET VAR', 'ups': 'Z3_SERVER', 'value': 'OL', 'error': ''}

        *** getUPSNumLogin
        {'cmd': 'NUMLOGINS', 'data': '2', 'error': ''}
        Terminated
         --- Internal NUTSocketClient deleted ---
```

Creating new Driver device class

```
class My_New_Device(DeviceBase):  # simply new class declaration
    def checkInputVoltage(self):  # overwrite methodes class
    ......
```

Data format who must be return by methods

By calling source class method `self.checkStatus()` you get DATA_TYPE_RETURN value. DATA_TYPE_RETURN is a dict type with keys.

```
DATA_TYPE_RETURN :
{
    'Modify' : True/False, # if the value change you must set to True.
    'xPLData' : {  # Here all value who must transmit to xPL message.
        'status' : <'mains' or 'battery', or 'unknown'>
        'event' : <All event define in XPL_Events>
    }
}
```

Use `self.handleXplEvent` to set XPL_Events status at True/False.

This method :

- Return a tuple with a boolean and a string for event key of DATA_TYPE_RETURN.

- Manage an XPL_Event retention for some XPL Events values.
- Ret others depending XPL_Events status of new event, Ex:

  if input_voltage_low' set to True

  'input_voltage_high' must set to False

  'input_voltage_ok' must set to False

How to use it :

```
....
retval = self.checkStatus()
retVal['modify'], retVal['xPLData']['event'] = self.handleXPL_Events(< an XPL_Events>, True/Fals
```

Dependency XPL_Events status

XPL_Events are groups in dependencies status, if an item change others items must change. Developer has no need to handle dependencies, `self.handleXplEvent` method handle them. It's just for information.

- **On line status:** 'onmains' => The UPS has begun operating on mains power

  'onbattery' => The UPS has begun operating on battery power
- **Battery status:** 'battlow' => The UPS battery is low

  'battfull' => The UPS battery is fully charged
- **Battery test status:** 'bti' => Battery test initiated

  'btp' => Battery Test Passed

  'btf' => Battery Test Failed
- **UPS server communucation status:** 'comms_lost' => The host has lost communication with the UPS

  'comms_ok' => Communication with the UPS has been restored
- **Input frequency status:** 'input_freq_error' => The input frequency is out of range

  'input_freq_ok' => The input frequency has returned from an error condition
- **Input voltage status:** 'input_voltage_high' => The input voltage is too high

  'input_voltage_low' => The input voltage is too low

  'input_voltage_ok' => The input voltage is OK following a previously "too low" or "too high" state
- **Output voltage status:** 'output_voltage_high' => The UPS output voltage is too high

  'output_voltage_low' => THe UPS output voltage is too low

  'output_voltage_ok' => The UPS output voltage has returned to normal following a "too high" or "too low" condition.
- **Output status:** 'output_overload' => The UPS output is in overload

  'output_ok' => The UPS output has returned from overload
- **Temperature status:** 'temp_high' => The UPS temperature is too high

  'temp_ok' => The UPS temperature has returned from an over-temperature condition

Overwrited methods

- **getBatteryCharge:** You can overwrite this method but it is essential to call source method at first, Because if NUT server has is own level it's probably the better.

  Overwriting structure:

```
        def getBatteryCharge(self):
            charge = DeviceBase.getBatteryCharge(self)
            if charge: return charge
              #.... Your new code in case of none 'battery.charge' handle by NUT server ....
            return charge # type float
```

- **checkAll:** Check All UPS stuff and return they values in DATA_TYPE_RETURN list.

  This method work with DeviceBase class, overwrite it only if new check method is necessary. Otherwise overwrite existing methods.

  You can overwrite this method but it is essential to call source method at first.

  Overwrite could be necessary only for new check, otherwise overwrite existing methods.

  Overwriting structure:

```
        def checkAll(self):
            data = DeviceBase.checkAll(self)
            data = data.append(self.Your_New_Check())
            .....
            return data
```

- **checkInputVoltage:** Source method do nothing and return None. You must overwrite it with status calculate to return event `input_voltage_low`, `input_voltage_high`, `input_voltage_ok` in format DATA_TYPE_RETURN

  Overwriting example:

```
        def checkInputVoltage(self):
              # Check if key exist in UPS vars
            if self._vars.has_key('input.voltage.nominal') :
                  # Calculat range voltage status
                high = self._vars['input.voltage.nominal'] * 1.06
                low = self._vars['input.voltage.nominal'] * 0.94
            else : # Return None to report not handling UPS event
                return None
              # Check if key exist in UPS vars
            if self._vars.has_key('input.voltage') :
                  # Get UPS status in DATA_TYPE_RETURN format
                retVal = self.checkStatus()
                  # Test voltage status
                if self._vars['input.voltage'] >= high :
                      # Set 'Modify' and XPL_Events DATA_TYPE_RETURN format
                    retVal['modify'], retVal['xPLData']['event'] = self.handleXPL_Events('input_volt
                elif self._vars['input.voltage'] <= low :
                        # Set 'Modify' and XPL_Events DATA_TYPE_RETURN format
                    retVal['modify'], retVal['xPLData']['event'] = self.handleXPL_Events('input_volt
                else :
                        # Set 'Modify' and XPL_Events DATA_TYPE_RETURN format
                    retVal['modify'], retVal['xPLData']['event'] = self.handleXPL_Events('input_volt
                return retVal # Return DATA_TYPE_RETURN format
            return None # Return None to report not handling UPS event
```

- **checkInputFreq:** Source method do nothing and return None. You must overwrite it with status calculate to return event `input_freq_error`, `input_freq_ok` in format DATA_TYPE_RETURN

  Overwriting structure : same principle than checkInputVoltage

- **checkOutputVoltage:** Source method do nothing and return None. You must overwrite it with status calculate to return event `ouput_voltage_low`, `output_voltage_high`, `output_voltage_ok` in format DATA_TYPE_RETURN

  Overwriting structure : same principle than checkInputVoltage

- **checkOutput:** Source method do nothing and return None. You must overwrite it with status calculate to return event `output_overload`, `output_ok` in format DATA_TYPE_RETURN

  Overwriting structure : same principle than checkInputVoltage

- **checkTemperature:** Source method do nothing and return None. You must overwrite it with status calculate to return event `temp_high`, `temp_ok` in format DATA_TYPE_RETURN

  Overwriting structure : same principle than checkInputVoltage

CHAPTER 3

Changelog

3.1 0.1a

- Plugin creation