webdyn
Flexitron group

WebdynSunPM
Application Note

LUA User Manual

Table of contents

LUA User Manual

1 Introduction

WebdynSunPM integrates a Lua scripting engine (V5.3) which allows advanced end users to run their own algorithms locally.

Lua is the programming language used by the scripting engine to run user scripts.

Lua is a complete scripting language. A LUA programming manual can be found at https://www.lua.org

> **Tip**
>
> Knowledge of the LUA language is a prerequisite for this user manual.

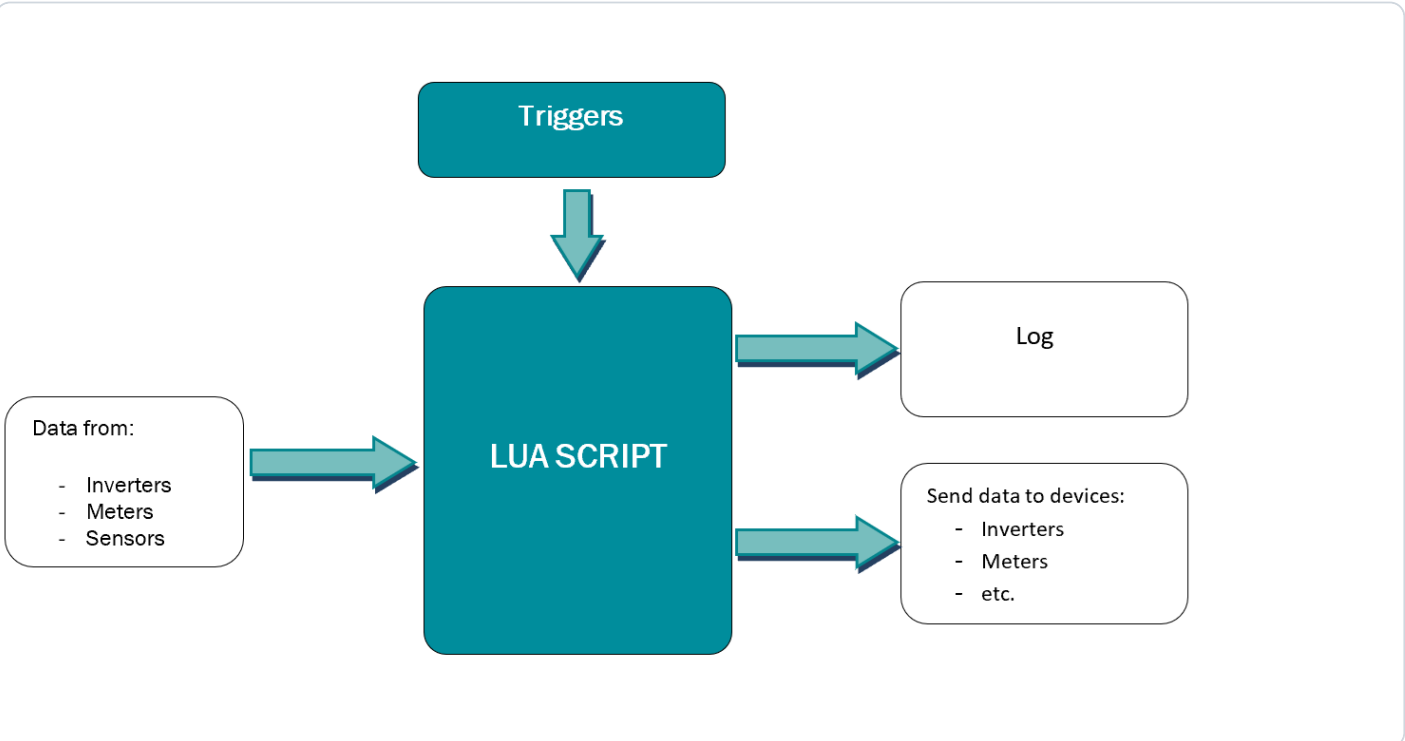
The Lua language, like many other languages, introduces the concept of function. The scripting engine integrated in the WebdynSunPM software will call these functions depending on the occurrence of certain events.

These events are called **triggers**.

When a function is triggered by an event, the code is executed. During this execution, no more events can be triggered: the LUA engine is single-threaded.

Pending events are queued in a FIFO but only one occurrence of a specific event will be kept.

For example, if a trigger is defined on a change of PWR variable of an inverter, at the first occurrence of the change, the associated function will be executed. If, when executing the function two other changes occur on the same variable, at the end of the execution of the function, function will be triggered once (and only once) more.

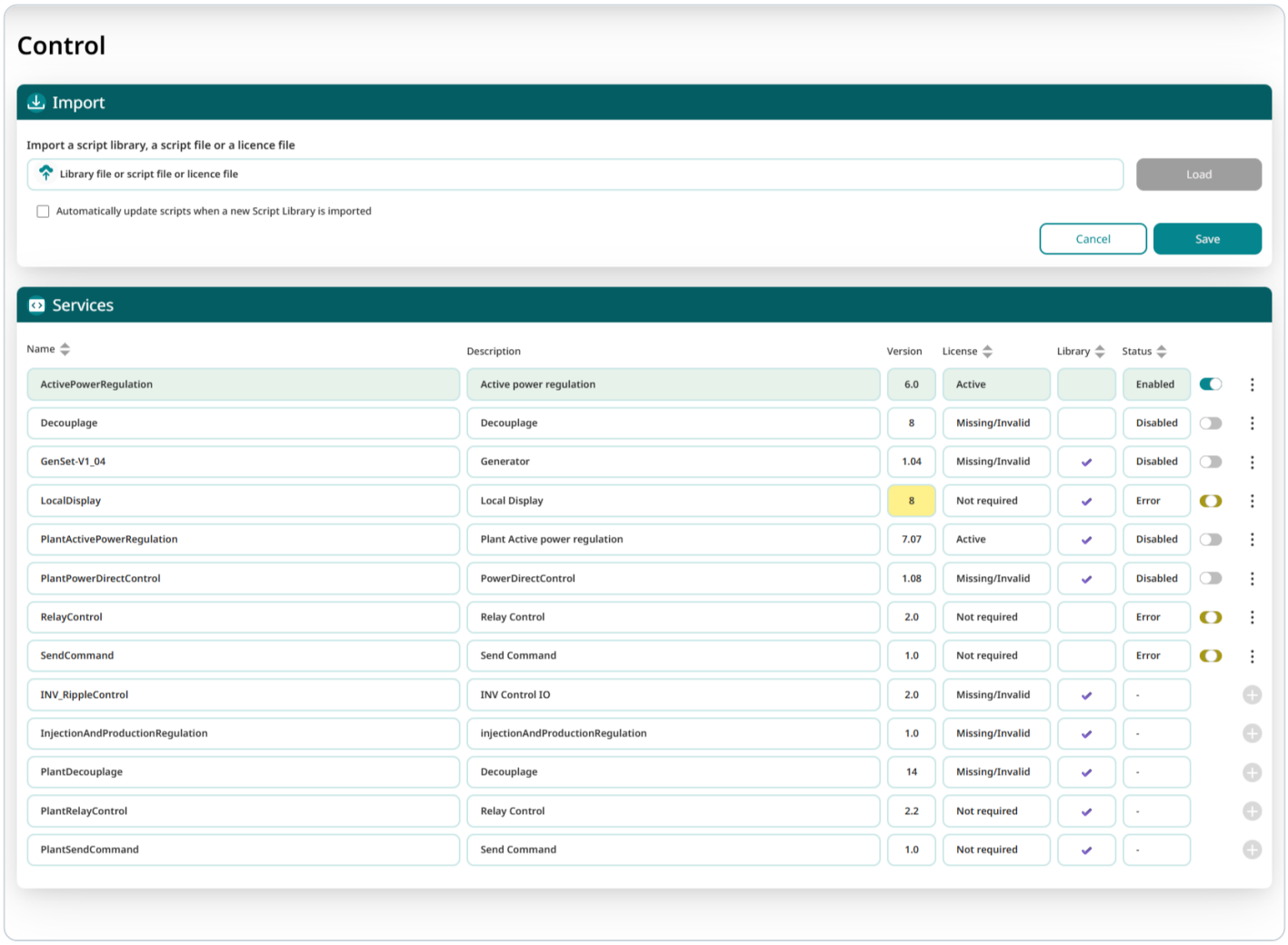


The WebdynSunPM provides a software development kit (SDK) with some functions to read data from devices (e.g. inverters, meters, etc.) and write data to devices.

This manual aims to describe the Lua programming language for WebdynSun PM, for script management with local web interface or remote server check the user manual.

2 Import a Script

2.1 Embedded web interface



control page

Go to the **Control** page then, click on the **Add script/licence file** button, select the Lua file to import and click on the import logo.

Upon successful import, the Lua script will be displayed in the table below.

2.2 Remote server

Put the script file in Lua format in the **SCRIPT** directory and add the new Lua file in the **<uid>_scl.ini** file in the **CONFIG** directory. (See in the WebdynSunPM manual)

> **Note**
>
> Lua is a scripting language, only the syntax is checked when importing it, the language is interpreted at runtime, missing variables, missing function, division by 0, etc. are only detected at runtime. In this case, the script is immediately stopped.

3 Script Lifecyle API

3.1 File Design

A Lua file consists of 2 parts which are:

Header:

The header must be present at the beginning of the file. The format is as follows:

```
header = {
version=1.0,
label=**testscript**
}
```

On the embedded web interface, the following fields correspond to:

- **version**: script version number
- **label**: to the brief description of the script

Functions:

A script can contain functions in the following format:

```
function my function ()
    return 1
end
```

According to the LUA language specification, a specific notation is adopted for function parameters and results.

Parameters:

Each function can accept zero or more parameters. Parameter types may or may not be known. Parameters are described as <type> name. If the type is not known, <?> will be used. If the parameter is optional, [<type> name] will be used.

Results:

Functions may return zero or more results. If a name is necessary for the explanation, it can be indicated in italics. The name of the result is only present for understanding since it has no real existence. If the result is optional then [<type> name] will be used. Functions returning no parameters will be described as <>.

The webdynSunPM has reserved functions which are part of the API and which are called when the conditions are met. These functions are:

- **wsInit()**: Called during script initialization
- **wsTick()**: Serving as a timer for periodic task execution.
- **wsStop()**: Called when stopping the script

3.2 `wsInit` : Script initialization

`[<string> error] wsInit([<string> parameters])`

Function called by the WebdynSunPM when the script is activated

Parameters

Name	Description
parameters	Parameters passed by the **Script args** field in the web interface or the **SCRIPT_Args[]** variable in the file <uid>_scl.ini (optional)

Results

Name	Description
Error	Text message explaining the error

When the script is activated, this function is automatically called without parameters.

This method is called on the following events:

- WebdynSunPM starts and the script is functional
- The user activated the script manually.

If an error is reported, the script stops and displays the error.

Example init script

```
function wsInit()
    local val = 1
    wd.log(**Start Init**)
    if val < 0 then
        error **value problem**
    end
    wd.log(**Init OK**)
end
```

Example log

```
2022-03-17 14:47:35 [test.lua 11] Start Init
2022-03-17 14:47:35 [test.lua 19] Init OK
```

Example init script with one parameter paramUser

```
function wsInit(param)
    wd.log(**Start Init**)
    if param == nil then
        wd.log**(**No parameter**)
    else
        wd.log(**parameter=**..param)
    end
    wd.log(**Init OK**)
end
```

Example log

```
2022-03-17 14:47:35 [test.lua 11] Start Init
2022-03-17 14:47:35 [test.lua 16] parameter=paramUser
2022-03-17 14:47:35 [test.lua 19] Init OK
```

3.3 `wsTick` : Timer (tick every second)

`<> wsTick()`

Function called by the WebdynSunPM every second if the script is enabled.

Parameters

None

Results

None

When the script is enabled, this function is automatically called every second without parameters.

Example timer script

```
local time_hour = 0
local time_min = 0
local time_sec = 0

function wsTick()
    time_sec = time_sec + 1
    if time_sec >= 60 then
        time_sec = 0
        time_min = time_min + 1
    end

    if time_min >= 60 then
        time_min = 0
        time_hour = time_hour + 1
    end

    if time_hour >= 24 then
        time_hour = 0
    end
    wd.log(**time=**..time_hour..**h**..time_min..**m**..time_sec..**s**)
end
```

Example log

```
2022-03-17 14:47:36 [test.lua 49] time=0h0m1s
2022-03-17 14:47:37 [test.lua 49] time=0h0m2s
2022-03-17 14:47:38 [test.lua 49] time=0h0m3s
2022-03-17 14:47:40 [test.lua 49] time=0h0m4s
2022-03-17 14:47:41 [test.lua 49] time=0h0m5s
2022-03-17 14:47:42 [test.lua 49] time=0h0m6s
```

3.4 `wsStop` : Stopping the script

`<> wsStop()`

Function called by the WebdynSunPM when the script is going to be stopped.

Parameters

None

Results

None

When the script is going to stop, this function is automatically called without parameter.

This function is called on the following events:

- A restart of the WebdynSunPM
- The script is disabled by the user
- The script is removed from the WebdynSunPM
- An error occurs in a LUA function (except in the wsStop function to avoid recursive calls)

Example of stopping the script

```
function wsStop()
    inverterOnOff(0) -- call of a function to turn off the inverters
    wd.log(**script stopped**)
end
```

Example log

```
2022-03-17 14:47:36 [test.lua 55] script stopped
```

4 WebdynSunPM API

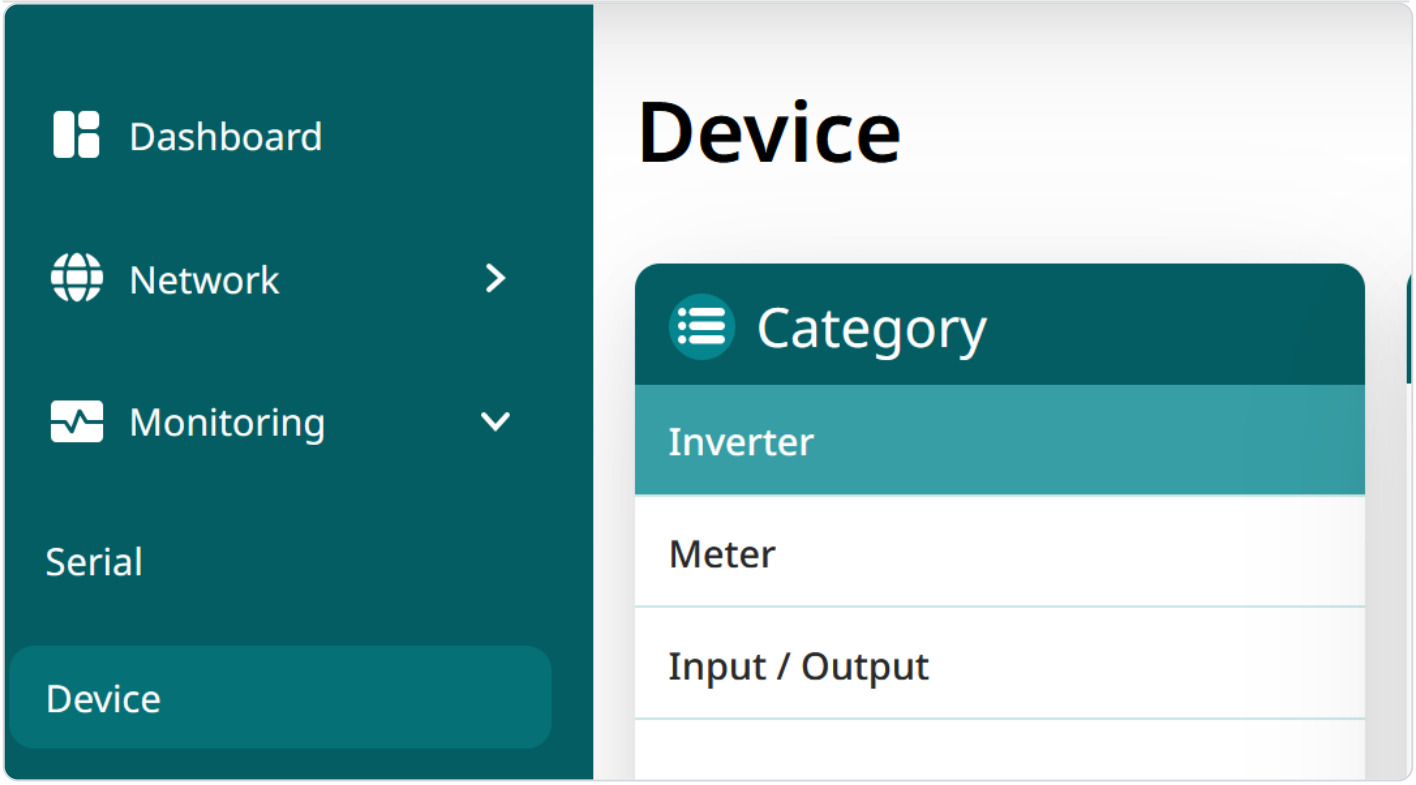
Many APIs have been implemented in the WebdynSunPM allowing communication with the various devices configured on it. All the variables of a device are represented as a Lua object. To access a variable, it must first be retrieved and then it becomes usable in the script.

4.1 Device Category

Categories identify a group of devices that has the same functional characteristics, for example inverters, meters, sensors, etc.

A functional characteristic is identified by a set of variables generally available in all equipment of the same category.

On the embedded web interface, they can be identified in the **Device page**



In the equipment definition file `\<uid>\_daq.csv`, the category is part of the parameters of each equipment.

> **Note**
>
> See WebdynSunPM User Manual

4.1.1 getCategoryList : List of categories

`<string[]>list, [<string>error] wd.getCategoryList()`

The Function allows to retrieve the list of category names on the WebdynSunPM

Parameters

None

Results

Name	Description
list	A table representing the list of category names as they appear in web pages

Example of display management for a category list

```
function CategoryListView()
    local i
    categoryList=wd.getCategoryList()
    if categoryList ~= nil then
        wd.log(**list:**)
        for i = 0, #listcat do
            wd.log(..category list[i])
        end
    else
```

```
        wd.log(**list is empty**)
    end
end
```

Example log

```
2022-03-17 15:19:12 [test.lua 29] list :
2022-03-17 15:19:12 [test.lua 29] Inverter
2022-03-17 15:19:12 [test.lua 29] Meter
2022-03-17 15:19:12 [test.lua 29] WebdynSunPM
```

4.1.2 getCategoryVars : Variable in a category

`<variable[]> variables wd.getCategoryVars(<string>CategoryName, <string>tagName)`

The function is used to retrieve all the values of a variable on all the devices of the same category.

Parameters

Name	Description
Category name	Device name as it appears in web pages.
tagName	Variable tag name

Results

Name	Description
vars	A variable array with all the variables corresponding to the parameters: If no variable matches the parameters, the array is empty.

The variable tag name **TagName** is the name of the **Tag** which must be added to the definition files of each device to be taken into account.

> **Note**
>
> See WebdynSunPM User Manual

Example of cumulating of a categorical variable

```
function GetGeneralPower()
    local PacTotal = 0

    PoweracVars = wd.getCategoryVars(**Inverter**, **Watts**)              -- Inverters power (w)
    NbInverter = #PoweracVars

    if PoweracVars ~= nil then
                    for i = 1, NbInverter do
                                if ((type(PoweracVars[i]) == **table**) and (PoweracVars[i].get()) ~=
nil) then
                    wd.log(**INV[**..i..**]=**..PoweracVars[i].get())
                                            PacTotal = PacTotal + PoweracVars[i].get()
                                else
                                            wd.log(**INV[**..i..**]=null**)
                                end
```

```
                        end
                        wd.log(**PacTotal(**..NbInverter..** Inverters)=**..PacTotal.. **W**)
                else
                        wd.log(**no variables in the category**)
                end
end
```

Example log

```
2022-03-17 17:31:48 [test.lua 85] INV[1]=1300
2022-03-17 17:31:48 [test.lua 82] INV[2]=1352
2022-03-17 17:31:48 [test.lua 85] INV[3]=null
2022-03-17 17:31:48 [test.lua 85] INV[4]=2674
2022-03-17 17:31:48 [test.lua 82] INV[5]=3658
2022-03-17 17:31:48 [test.lua 85] INV[6]=1670
2022-03-17 17:31:48 [test.lua 85] INV[7]=5687
2022-03-17 17:31:48 [test.lua 88] PacTotal(7 Inverters)=16341W
```

4.1.3 getCategoryTagVars : Variable in a category for tagged devices

```
<variable[]> variables wd.getCategoryTagsVars(<string>CategoryName, <string>deviceTagName, <string>t
agName)
```

The function is used to retrieve all the values of a variable on all the devices of the same category with a given deviceTag

Parameters

Name	Description
Category name	Device name as it appears in web pages.
deviceTagName	Device tagName : to remove not use in the next version
tagName	Variable tag name

Results

Name	Description
vars	A variable array with all the variables corresponding to the parameters:If no variable matches the parameters, the array is empty.

The variable tag name **TagName** is the name of the **Tag** which must be added to the definition files of each device to be taken into account.

Example of cumulating of a categorical variable

```
function GetGeneralPower()
        local PacTotal = 0

        PoweracVars = wd.getCategoryTagVars(**Inverter**, **MainSolarString**,
**Watts**)                -- Inverters power (w)
        NbInverter = #PoweracVars

        if PoweracVars ~= nil then
```

```
                for i = 1, NbInverter do
                        if ((type(PoweracVars[i]) == **table**) and (PoweracVars[i].get()) ~= nil) t
hen
                                wd.log(**INV[**..i..**]=**..PoweracVars[i].get())
                                PacTotal = PacTotal + PoweracVars[i].get()
                        else
                                wd.log(**INV[**..i..**]=null**)
                        end
                end
                wd.log(**PacTotal(**..NbInverter..** Inverters)=**..PacTotal.. **W**)
        else
                wd.log(**no variables in the category**)
        end
end
```

Example log

```
2022-03-17 17:31:48 [test.lua 85] INV[1]=1300
2022-03-17 17:31:48 [test.lua 82] INV[2]=1352
2022-03-17 17:31:48 [test.lua 85] INV[3]=null
2022-03-17 17:31:48 [test.lua 85] INV[4]=2674
2022-03-17 17:31:48 [test.lua 82] INV[5]=3658
2022-03-17 17:31:48 [test.lua 85] INV[6]=1670
2022-03-17 17:31:48 [test.lua 85] INV[7]=5687
2022-03-17 17:31:48 [test.lua 88] PacTotal(7 Inverters)=16341W
```

4.1.4 getCategoryVar : First variable in a category

`<variable> variables wd.getCategoryVars(<string>CategoryName, <string>tagName)`

The function is used to retrieve the first values of a variable on the first devices of the same category.

Parameters

Name	Description
Category name	Category name as it appears in web pages
tagName	Variable tag name

Results

Name	Description
var	A variable with the first variables corresponding to the parameters: If no variable matches the parameters, the variable is empty.

The device tag name **deviceTagName** is the device tag which must be added to the device in DAQ file. (see user manual) The variable tag name **TagName** is the name of the **Tag** which must be added to the definition files of each device to be taken into account.

4.1.5 getCategoryTagVar : The first Variable in a category for tagged devices

DEPRECATED : DO NOT USE

`<variable> variables wd.getCategoryVar(<string>CategoryName, <string>deviceTagName, <string>tagName)`

The function is used to retrieve the first value of a variable on the first device with the given category with a given deviceTag

Parameters

Name	Description
Category name	Category name as it appears in web pages.
deviceTagName	Device tagName
tagName	Variable tag name

Results

Name	Description
var	A variable with the first variable corresponding to the parameters:If no variable matches the parameters, the variable is empty

The device tag name **deviceTagName** is the device tag which must be added to the device in DAQ file. (see user manual) The variable tag name **TagName** is the name of the **Tag** which must be added to the definition files of each device to be taken into account.

4.1.6 onCategoryVarChange : Function for triggering a tag on an category

```
<> wd.onCategoryVarChange(<string>CategoryName, <string>tagName, <function>callback)
```

The function allows you to monitor the changes of the variable on the whole category and to trigger the execution of a specific function in the event of a change in value.

Parameters

Name	Description
Category name	Category name
tagName	Variable tag name to monitor
callback	Function to call when a variable corresponding to the parameters will change.

Results

None

The variable tag name **TagName** is the name of the **Tag** which must be added to the definition files of each device to be taken into account.

> **Warning**
>
> The **wd.onCategoryVarChange** function systematically triggers the callback function at the startup, even before the first change

Example

Example of instant accumulation following a change in a category variable

```
function wsInit()
        local val = 1

        wd.log(**Start Init**)

        wd.onCategoryVarChange(**Inverter**, **Watts**, GetGeneralPower)

        wd.log(**Init OK**)
end

function GetGeneralPower()
        local PacTotal = 0

        PoweracVars = wd.getCategoryVars(**Inverter**, **Watts**)               -- Inverters power
(w)
        NbInverter = #PoweracVars

        if PoweracVars ~= nil then
                for i = 1, NbInverter do
                        if ((type(PoweracVars[i]) == **table**) and (PoweracVars[i].get()) ~= nil) t
hen
                                wd.log(**INV[**..i..**]=**..PoweracVars[i].get())
                                PacTotal = PacTotal + PoweracVars[i].get()
                        else
                                wd.log(**INV[**..i..**]=null**)
                        end
                end
                wd.log(**PacTotal(**..NbInverter..** Inverters)=**..PacTotal.. **W**)
        else
                wd.log(**no variables in the category**)
        end
        return PacTotal
end
```

Example log

```
2022-03-18 10:41:19 [test.lua 12] Start Init
2022-03-18 10:41:19 [test.lua 20] Init OK
2022-03-18 10:41:19 [test.lua 85] INV[1]=1300
2022-03-18 10:41:19 [test.lua 82] INV[2]=1352
2022-03-18 10:41:19 [test.lua 85] INV[3]=3562
2022-03-18 10:41:19 [test.lua 88] PacTotal(3 Inverters)=6214W
2022-03-18 10:42:00 [test.lua 85] INV[1]=2694
2022-03-18 10:42:00 [test.lua 82] INV[2]=1352
2022-03-18 10:42:00 [test.lua 85] INV[3]=3562
2022-03-18 10:42:00 [test.lua 88] PacTotal(3 Inverters)=7608W
2022-03-18 10:43:37 [test.lua 85] INV[1]=2694
2022-03-18 10:43:37 [test.lua 82] INV[2]=4512
2022-03-18 10:43:37 [test.lua 85] INV[3]=2857
2022-03-18 0:43:37 [test.lua 88] PacTotal(3 Inverters)=10063W
```

4.2 Device

A device is an equiment connected to the WebdynSunPM. The inputs/outputs of the WebdynSunPM are also considered as device as well as virtual devices created by scripts. A device has a unique name that identifies it, for virtual devices, the device name is the script name. For connected devices, this name is editable and visible on the embedded web interface in **Devices**:

This name is also available under the **name** parameter in the **<uid>_daq.csv** configuration file available in the CONFIG directory.

Each device is part of a category and presents a set of variables that can be read or possibly written. A variable must have its **Tag** parameter filled in to be used by the script. The list of variables is available in the definition files of each device.

4.2.1 getDeviceStatus : Equipment status

```
<variable> state wd.getDeviceStatus(<string> deviceName)
```

The function allows the state of a device

Parameters

Name	Description
deviceName	Equipment name

Results

State	Equipement status:
none	unknown status (grey on the web)
1	The equipment has been found and the current configuration is functional (green on web)
2	The equipment has been found but one or more variables in the definition file are not functional (orange on the web)
3	The device was not found or the current configuration is not functional (Red on the web)

Example

Example of retrieving the state of a device

```
function GetStateInverterSimu()
        state = wd.getDeviceStatus(**Inverter_simu**)

        if state ~= nil then
                wd.log(**Inverter state = error**)
        elseif state == 0 then
                wd.log(**Inverter state = status unknown**)
        elseif state == 1 then
                wd.log(**Inverter state = device OK**)
        elseif state == 2 then
                wd.log(**Inverter state = device warning**)
        elseif state == 3 then
                wd.log(**Inverter state = device error**)
        end
end
```

Example log

```
2022-03-18 14:44:20 [test.lua 92] Inverter state = device OK
```

4.2.2 getDeviceList : list all devices in a category

```
<string[ ]> variables wd.getDeviceList(<string>CategoryName)
```

The function is used to retrieve all the device name on all the devices of the same category.

Parameters

Name	Description
Category name	Category name as it appears in web pages.

Results

Name	Description
list	A string array with all the devices name in the category. If no device is in the category, the array is empty.

Example

Example of cumulating of a categorical variable

```
function inverterList()

        inverterList = wd.getDeviceList(**Inverter**)
        NbInverter = #inverterList
        for i = 1, NbInverter do
                wd.log(**INV[**..i..**]=** .. inverterList[i])
        end
end
```

Example log

```
2022-03-17 17:31:48 [test.lua 85] INV[1]=Inverter 1
2022-03-17 17:31:48 [test.lua 82] INV[2]=Inverter 2
```

4.2.3 checkDeviceVar : Check equipment variable

`<variable> var wd.checkDeviceVar(<string> deviceName , <string> tagName )`

The function is used check if a particular device has a particular tag

Parameters

Name	Description
Equipment name	Equipment name
TagName	Variable tag name

Results

Name	Description
isError	true: the device exist and the tag is included in its definition file
errorMessage (if isError=true)	Explicit error message (Device not found OR Tag not found)

The variable tag name **TagName** is the name of the **Tag** which must be added to the definition files of each device to be taken into account.

Example

Example of retrieving the value of an equipment variable:

```
function checkVariable()
        isError, errorMessage = wd.checkDeviceVar(**Inverter_simu**, **Info**)

        if isError then
                wd.log(**Info Inverter Simu=**.. errorMessage)
        else
                wd.log(**variable exists**)
        end
end
```

Example log

```
2022-03-18 14:44:20 [test.lua 107] Info Inverter Simu=Device not found
```

4.2.4 getDeviceVar : Equipment variable

`<variable> var wd.getDeviceVar(<string> deviceName , <string> tagName )`

The function is used to retrieve the value of a variable from a device

Parameters

Name	Description
Equipment name	Equipment name
TagName	Variable tag name

Results

Name	Description
var	A variable corresponding to the parameters:If only one matches the parameters, it will be returned. If no variable matches the parameters, **nil** is returned.

The variable tag name **TagName** is the name of the **Tag** which must be added to the definition files of each device to be taken into account.

Example of retrieving the value of an equipment variable:

```
function GetInfoInverterSimu()
        InfoInverterSimu = wd.getDeviceVar(**Inverter_simu**, **Info**)

        if InfoInverterSimu.get() ~= nil then
                wd.log(**Info Inverter Simu=**.. InfoInverterSimu.get())
        else
                wd.log(**no variable**)
        end
end
```

Example log:

```
2022-03-18 14:44:20 [test.lua 107] Info Inverter Simu=test
```

4.2.5 onDeviceVarChange : Function for triggering a tag on an equipment

`wd.onDeviceVarChange(<string>deviceName, <string>tagName, <function>callback)`

The scripted function sets a trigger on variable changes.

Parameters

Name	Description
Equipment name	Equipment name
tagName	Variable tag name to monitor
Callback	Function called when the monitored variable has changed

Results

None

The variable tag name **TagName** is the name of the **Tag** which must be added to the definition files of each device to be taken into account.

> **Warning**
>
> The **wd.onDeviceVarChange function systematically triggers the callback function at the startup, even before the first change

Example of display of the last modified value of a device variable:

```
function wsInit()
        local val = 1

        wd.log(**Start Init**)

        wd.onDeviceVarChange(**Inv_1**, **Watts**, GetInverterPower)

        wd.log(**Init OK**)
end

function GetInverterPower ()
        InfoInverterSimu = wd.getDeviceVar(**Inv_1**, **Watts**)

        if InfoInverterSimu.get() ~= nil then
                wd.log(**New value Inv_1=**..InfoInverterSimu.get()..**W**)
        else
                wd.log(**no variable**)
        end
end
```

Example log

```
2022-03-18 10:41:19 [test.lua 12] Start Init
2022-03-18 10:41:19 [test.lua 20] Init OK
2022-03-18 10:41:19 [test.lua 85] New value Inv_1=1750W
2022-03-18 10:45:36 [test.lua 85] New value Inv_1=2354W
2022-03-18 10:53:42 [test.lua 85] New value Inv_1=3498W
```

4.3 Variable object

An object variable must be attached to a variable of a device or a category. It is then possible to read or write to this object variable.

4.3.1 Reading an object variable

`<?> value variable.get()`

Function that returns the last known value of a variable.

Parameters

None

Results

Name	Description
Value	Value of the variable, i.e. the last value as displayed on the website. The type depends on the type of value

Example of reading a variable:

```
function GetInverterPower ()
        InfoInverterSimu = wd.getDeviceVar(**Inv1**, **Watts**)

        if InfoInverterSimu.get() ~= nil then
                wd.log(**New value Inv1=**..InfoInverterSimu.get()..**W**)
        else
                wd.log(**no variable**)
        end
end
```

Example log

```
2022-03-18 10:41:19 [test.lua 85] New value Inv1=1750W
```

4.3.2 Synchronized reading an object variable

`<?> value variable.getSync()`

Function that returns the value of a variable after performing a physical reading on the device. The function is blocked until the end of the reading on the equipment.

Parameters

None

Results

Name	Description
Value	Value of the variable. The type depends on the type of value

Example

Example of reading a variable:

```
function GetInverterPower ()
        local val
        InfoInverterSimu = wd.getDeviceVar(**Inv1**, **Watts**)

        val = InfoInverterSimu.getSync()
        if val ~= nil then
                wd.log(**New value Inv1=**..val..**W**)
        else
                wd.log(**no variable**)
        end
end
```

Example log

```
2022-03-18 10:41:19 [test.lua 85] New value Inv1=1750W
```

4.3.3 Reading an object variable with timestamp information

`<structure> info variable.getEx()`

Function that returns the last known value of a variable with the timestamp of the last acquisition and the number of failed reads since last read.

Parameters

None

Results

- **Info**:
  - value : Value of the variable. The type depends on the type of value
  - timestamp : Timestamp at the data acquisition point. Number of milliseconds elapsed since January 1, 1970
  - errorCount : Number of failed reads since last read

Example of reading a variable:

```
function GetInverterPower ()
        local val
        InfoInverterSimu = wd.getDeviceVar(**Inv**, **Watts**)

        val = InfoInverterSimu.getEx()
        if val.value ~= nil then
                wd.log(**Value Inv = **..val.value..**W**)
                wd.log(**Acquisition timestamp = **..val.timeStamp..**ms**)
                wd.log(**Count of errors since this timestamp = **..val.errorCount)
        else
                wd.log(**no variable**)
```

```
        end
end
```

Example log

```
2022-05-30 10:41:19 [test.lua 82] Value Inv = 2306W
2022-05-30 10:41:19 [test.lua 83] Acquisition timestamp = 1653900020000ms
2022-05-30 10:41:19 [test.lua 84] Count of errors since this timestamp = 0
```

4.3.4 Writing an object variable

`<Boolean> result , [<string> message] variable.set( <?> value)`

Function that allows you to modify the value of a variable.

Parameters

Name	Description
Value	Value of the variable to define on the device. The variable must be read/write (i.e. the device must be able to physically accept the change)

Results

Name	Description
Flag	true: The result is OK, the variable as written
	false: A problem occurs, the variable was not written, the error message can be used as an error.
Error	Text message explaining the error

Example of writing a variable:

```
function SetInverterCoef(value)
        CoefInverterSimu = wd.getDeviceVar(**Inv1**, **Coef**)

        result=CoefInverterSimu.set(value)
        if result == true then
                wd.log(**variable set to**..value)
        else
                wd.log(**variable setting error**)
        end

end
```

Example log

```
2022-03-21 08:44:01 [test.lua 115] variable set to 23
```

4.3.5 Unsynchronized Writing an object variable

`<Boolean> result , [<string> message] variable.setAsync( <?> value)`

Function that allows you to modify the value of a variable. The command is queued to the inverter and the function return without waiting for the result.

Parameters

Name	Description
Value	Value of the variable to define on the device. The variable must be read/write (i.e. the device must be able to physically accept the change)

Results

Name	Description
Flag	**true**: The result is OK, but no guarantee that the variable as been written
	false: A problem occurs, the variable was not written, the error message can be used as an error.
Error	Text message explaining the error

Example of writing a variable:

```
function SetInverterCoef(value)
        CoefInverterSimu = wd.getDeviceVar(**Inv1**, **Coef**)

        result=CoefInverterSimu.setAsync(value)
        if result == true then
                wd.log(**queue variable set to**..value)
        else
                wd.log(**variable setting error**)
        end

end
```

Example log

```
2022-03-21 08:44:01 [test.lua 115] variable set to 23
```

4.3.6 RAW values handling

RAW values are the values which are identified as RAW in definition file. It represents a number of registers that should be read and written in one request with read/write block modbus function codes.

The field info2 in definition files represents <address>_<size> value where * Address is the modbus address of the first register * Size * 2 is the number of registers in the block For these values: * the type of the value returned from the get commands is an integer array. * The type of the value expected by the set commands is an integer array.

Block definition example

```
modbus;Inverter;Webdyn;BlockDevice
1;3;10000_40;RAW;;Block;Block;1.000000;0.000000;;4
```

script for reading and writing the block

```
Block = wd.getDeviceVar(**Inverter **,**Block**)

-- Display contents
-- Read the full block
readall=Block.get()
for k, v in pairs(readall) do
    wd.log(**block[**.. k .. **] = ** ..  v)
end

-- Increment contents
for k, v in pairs(readall) do
    readall[k] = v + 1
    if (readall[k] >= 65535) then
        readall[k] = 0
    end
end
-- Write the full block
Block.set(readall)
```

4.4 System

4.4.1 Log

`<> wd .log(<string> message)`

Function called by script log a line.

Parameters

Name	Description
Message	Message to write in the journal.

Results

None

Example of writing in the log:

```
function log()
        local I = 42
        wd.log(**writing in the log number**..i)
end
```

Example log :*

```
2022-03-21 08:44:01 [test.lua 115] writing in the log number:42
```

4.4.2 Save a configuration

`<> wd.save(<?> config)`

Save an object as script parameters

Parameters

Name	Description
Configuration	Configuration to save as script settings

Results

None

> **Warning**
>
> The configuration must be a lua Object

> **Warning**
>
> Deleting the script will also delete the configuration

Example of saving a configuration:

```
local default_config = {param1 = 0, param2 = 0, param3 = 0}

function wsInit()
        wd.log(**Start Init**)

        config = wd.load()

        if config ~= nil then
                wd.log(**Config load **..config.param1..** **..config.param2..** **..config.param3)
        else
                wd.log(**NO config load**)
                config = default_config
        end

        wd.log(**Init Ok**)
end

function wsStop()
        config.param1 = config.param1 + 1
        config.param2 = config.param2 + 2
        config.param3 = config.param3 + 5

        if config ~= nil then
                wd.log(**Config save **..config.param1..** **..config.param2..** **..config.param3)
                wd.save(config)
        else
                wd.log(**Config save error**)
        end

        wd.log(**script stopped**)
end
```

Example log

```
2022-03-21 13:56:15 [test.lua 12] Start Init
2022-03-21 13:56:15 [test.lua 19] NO config load
2022-03-21 13:56:15 [test.lua 23] Init OK
2022-03-21 13:56:17 [test.lua 36] Config save=1 2 5
2022-03-21 13:56:17 [test.lua 42] script stopped
2022-03-21 13:56:27 [test.lua 12] Start Init
```

```
2022-03-21 13:56:27 [test.lua 17] Config load=1.0 2.0 5.0
2022-03-21 13:56:27 [test.lua 23] Init OK
2022-03-21 13:56:28 [test.lua 36] Config save=2.0 4.0 10.0
2022-03-21 13:56:28 [test.lua 42] script stopped
2022-03-21 13:56:41 [test.lua 12] Start Init
2022-03-21 13:56:41 [test.lua 17] Config load=2.0 4.0 10.0
2022-03-21 13:56:41 [test.lua 23] Init OK
2022-03-21 13:56:42 [test.lua 36] Config save=3.0 6.0 15.0
2022-03-21 13:56:42 [test.lua 42] script stopped
2022-03-21 13:56:43 [test.lua 12] Start Init
2022-03-21 13:56:43 [test.lua 17] Config load=3.0 6.0 15.0
2022-03-21 13:56:43 [test.lua 23] Init OK
2022-03-21 13:57:03 [test.lua 36] Config save=4.0 8.0 20.0
2022-03-21 13:57:03 [test.lua 42] script stopped
```

4.4.3 Loading a configuration

`<config> wd.load()`

Loading script parameters into an object

Parameters

None

Results

Name	Description
Configuration	Configuration saved as script settings

> **Warning**
>
> The configuration must be a lua Object

> **Warning**
>
> Deleting the script will also delete the configuration

Example of loading a configuration:

```lua
local default_config = {param1 = 0, param2 = 0, param3 = 0}

function wsInit()
        wd.log(**Start Init**)

        config = wd.load()

        if config ~= nil then
                wd.log(**Config load **..config.param1..** **..config.param2..** **..config.param3)
        else
                wd.log(**NO config load**)
                config = default_config
        end

        wd.log(**Init OK**)
end
```

```
function wsStop()
        config.param1 = config.param1 + 1
        config.param2 = config.param2 + 2
        config.param3 = config.param3 + 5

        if config ~= nil then
                wd.log(**Config saved**..config.param1..** **..config.param2..** **..config.param3)
                wd.save(config)
        else
                wd.log(**Config save error**)
        end

        wd.log(**script stopped**)
end
```

Example log

```
2022-03-21 13:56:15 [test.lua 12] Start Init
2022-03-21 13:56:15 [test.lua 19] NO config load
2022-03-21 13:56:15 [test.lua 23] Init OK
2022-03-21 13:56:17 [test.lua 36] Config save=1 2 5
2022-03-21 13:56:17 [test.lua 42] script stopped
2022-03-21 13:56:27 [test.lua 12] Start Init
2022-03-21 13:56:27 [test.lua 17] Config load=1.0 2.0 5.0
2022-03-21 13:56:27 [test.lua 23] Init OK
2022-03-21 13:56:28 [test.lua 36] Config save=2.0 4.0 10.0
2022-03-21 13:56:28 [test.lua 42] script stopped
2022-03-21 13:56:41 [test.lua 12] Start Init
2022-03-21 13:56:41 [test.lua 17] Config load=2.0 4.0 10.0
2022-03-21 13:56:41 [test.lua 23] Init OK
2022-03-21 13:56:42 [test.lua 36] Config save=3.0 6.0 15.0
2022-03-21 13:56:42 [test.lua 42] script stopped
2022-03-21 13:56:43 [test.lua 12] Start Init
2022-03-21 13:56:43 [test.lua 17] Config load=3.0 6.0 15.0
2022-03-21 13:56:43 [test.lua 23] Init OK
2022-03-21 13:57:03 [test.lua 36] Config save=4.0 8.0 20.0
2022-03-21 13:57:03 [test.lua 42] script stopped
```

4.4.4 Delay Timer

`<> wd.sleep(<> time_ms)`

Wait time in milliseconds before the next instruction

Parameters

Name	Description
time_ms	Time in milliseconds

Results

None

Example of a 1500ms delay:

```
function tempo()
        wd.log(**Start of timer**)
        wd.sleep(1500)
```

```
        wd.log(**End of timer**)
end
```

Example log

```
2022-03-18 10:41:19 [test.lua 15] Start of timer
2022-03-18 10:41:20 [test.lua 17] End of timer
```

4.4.5 Send Alarms

`<> wd.sendAlarm(<string> alarmType, <string> msg, <Boolean> delayed)`

Send an alarm to the server

Parameters

Name	Description
alarmType	Alarm type in the alarm file
msg	Message for the alarm
Delayed	This parameter is optional **false**: the alarm is sent instantly (default)
	true: the alarm will be sent at the next connection

Results

None

Example of sending an alarm directly:

```
function triggerInstantAlarm()
        wd.sendAlarm(**AlarmType**,**AlarmName**, false)
end

function triggerDelayedAlarm()
        wd.sendAlarm(**AlarmType**,**AlarmName**, true)
end
```

Example alarm posted on FTP : ./ALARM/WPM_XXXXXX_AL_230110_120410.csv.gz

```
23/01/10-12:04:05;AlarmType;AlarmName
```

4.4.6 JSON parser

`<any> wd.json(<string> parameters)`

Allows you to retrieve the parameters in JSON format from the script

Parameters

Name	Description
Parameters	Parameters in JSON format retrieved in the script arguments

Results

any

Lua variable corresponding to the parameters in JSON format (nil if conversion error)

Example of init function with JSON parameters:

```
Function wsInit(jsonParameters)
        if (jsonParameters == nil) then
                wd.log(**Missing parameters, please edit settings**)
                error(**STOP**)
        end

        parameters = wd.json(jsonParameters)

        if (parameters == nil) then
                wd.log(**parameters not a valid JSON, please edit settings**)
                error(**STOP**)
        end


        wd.log(**parameter string=**..parameters.param1)
        wd.log(**parameter digital=**..parameters.param2)
        wd.log(**parameter boolean=**..parameters.param3)
end
```

4.5 Virtual device

From a Lua script, it is possible to create a virtual device. A script can declare variables on its virtual device, accessible for reading and writing.

The values of these variables are saved as a device in a data file, then deposited on the remote server. This virtual device has no physical existence and can only be used by its Lua script. The definition file is managed during the execution of the Lua script and deposited for informational purposes only on the remote server. No modification of the virtual device definition file is accepted by the concentrator.

The format of the virtual equipment definition file name is:

`\<uid>_SCRIPT_\<script_name>_\<script_version>.csv`

With : * `\<uid>` : Concentrator identifier * `\<script_name>` : Script file name * `\<script_version>` : Script version

Contents of the virtual device definition file generated by the Lua script : The file is in csv format, it is composed of text rows each composed of `;` delimited fields. The first row in the file contains the following information:

```
Protocol ; Category ; Manufacturer ; Model
```

The fields are configured as follows:

Field	Description
Protocol	Fixed value : `none`
Category	Fixed value : `Script`
Manufacturer	Fixed value : `Script`
Model	Lua script file name

The other lines will have the following format:

```
Index ; Info1 ; Info2 ; Info3 ; Info4 ; Name ; Tag ; CoefA ; CoefB ; Unit ; Action
```

The field meanings are the following:

Field	Description
Index	Contains the unique identifier of the variable in the file
Info1	None
Info2	None
Info3	Variable format. The allowed formats are:
	** U8:** 8-bit unsigned integer (1 byte)
	** U16:** 16-bit unsigned integer (2 bytes, or 1 register)
	U32: 32-bit unsigned integer (4 bytes, or 2 registers)
	U64: 64-bit unsigned integer (8 bytes, or 4 registers)
	I8: 8-bit signed integer (1 byte)
	I16: 16-bit signed integer (2 bytes, or 1 register)
	I32: 32-bit signed integer (4 bytes, or 2 registers)
	I64: 64-bit signed integer (8 bytes, or 4 registers)
	F32: 32-bit floating point (4 bytes, or 2 registers)
	F64: 64-bit floating point (8 bytes, or 4 registers)
	String: The variable is a character string
Info4	None
Name	Variable name in Lua script
Tag	Variable name in Lua script
CoefA	Fixed value : **1**
CoefB	Fixed value : **0**
Unit	None
Action	The only possible action code is 4. The variable is of instantaneous value type.

Note

Variables of a virtual device created by a Lua script can be accessed using the device's read and write functions.

4.5.1 Virtual Device Data File Format

The data file is made up of declared devices followed by virtual devices. Virtual devices have as header:

```
SCRIPTINDEX;NumScript_1
none;fileDefinitionName_1
...
...
SCRIPTINDEX;NumScript_N
none;fileDefinitionName_N
```

With:

- NumScript_N : Index of the script in the script configuration file `\<uid>_scl.ini`
- fileDefinitionName_N : Definition file name for virtual device N

The data has the same format as the other equipment.

4.5.2 declareScriptVars : Declare the variables of a virtual device

`<> wd.declareScriptVars (<?> var, <> AcqPeriod)`

Declare variables to the virtual equipment attached to this script

Parameters

Name	Description
var	Variables to attach to the virtual device. Variables are declared as an object
AcqPeriod	The acquisition period in seconds. (optional) Default : 600 seconds

var are declared as bellow

```
{
        <optional_type>_<nameVar> = x,
        <optional_type>_<nameVar> = x,
        <optional_type>_<nameVar> = x
}
```

`<optional_type>` are only numeric and strings :

- U8
- U16
- U32
- U64
- I8
- I16
- I32
- I64
- F32
- F64
- STRING

> **Note**
>
> If no optional type is specified, F64 will be used for numbers, STRING for strings.

> **Note**

<optional_type> is the name of the variable.

Results

None

Example of creating and using a virtual device:

```
declarations = {
        U8_var1 = 0,
        U16_var2 = 0,
        U32_var3 = 0,
        U64_var4 = 0,
        I8_var5 = 0,
        I16_var6 = 0,
        I32_var7 = 0,
        I64_var8 = 0,
        F32_var9 = 0,
        F64_var10 = 0,
        var11 = 0,
        STRING_var12 = **Test**
}

function wsInit()
        wd.log(**Start Init**)

        wd.declareScriptVars(declarations,60)

        local status, message = wd.checkDeviceVar(**TestVirtualDevice**,**var1**)
        if (status == false) then
                wd.log(**missing tag U8_var1**)
                isError = true
        else
                Var1 = wd.getDeviceVar(**TestVirtualDevice**,**var1**)
        end

        local status, message = wd.checkDeviceVar(**TestVirtualDevice**,**var2**)
        if (status == false) then
                wd.log(**missing tag U16_var2**)
                isError = true
        else
                Var2 = wd.getDeviceVar(**TestVirtualDevice**,**var2**)
        end

        local status, message = wd.checkDeviceVar(**TestVirtualDevice**,**var3**)
        if (status == false) then
                wd.log(**missing tag U32_var3**)
                isError = true
        else
                Var3 = wd.getDeviceVar(**TestVirtualDevice**,**var3**)
        end

        local status, message = wd.checkDeviceVar(**TestVirtualDevice**,**var4**)
        if (status == false) then
                wd.log(**missing tag U64_var4**)
                isError = true
        else
                Var4 = wd.getDeviceVar(**TestVirtualDevice**,**var4**)
        end

        local status, message = wd.checkDeviceVar(**TestVirtualDevice**,**var5**)
        if (status == false) then
                wd.log(**missing tag I8_var5**)
```

```
                isError = true
        else
                Var5 = wd.getDeviceVar(**TestVirtualDevice**,**var5**)
        end

        local status, message = wd.checkDeviceVar(**TestVirtualDevice**,**var6**)
        if (status == false) then
                wd.log(**missing tag I16_var6**)
                isError = true
        else
                Var6 = wd.getDeviceVar(**TestVirtualDevice**,**var6**)
        end

        local status, message = wd.checkDeviceVar(**TestVirtualDevice**,**var7**)
        if (status == false) then
                wd.log(**missing tag I32_var7**)
                isError = true
        else
                Var7 = wd.getDeviceVar(**TestVirtualDevice**,**var7**)
        end

        local status, message = wd.checkDeviceVar(**TestVirtualDevice**,**var8**)
        if (status == false) then
                wd.log(**missing tag I64_var8**)
                isError = true
        else
                Var8 = wd.getDeviceVar(**TestVirtualDevice**,**var8**)
        end

        local status, message = wd.checkDeviceVar(**TestVirtualDevice**,**var9**)
        if (status == false) then
                wd.log(**missing tag F32_var9**)
                isError = true
        else
                Var9 = wd.getDeviceVar(**TestVirtualDevice**,**var9**)
        end

        local status, message = wd.checkDeviceVar(**TestVirtualDevice**,**var10**)
        if (status == false) then
                wd.log(**missing tag F64_var10**)
                isError = true
        else
                Var10 = wd.getDeviceVar(**TestVirtualDevice**,**var10**)
        end

        local status, message = wd.checkDeviceVar(**TestVirtualDevice**,**var11**)
        if (status == false) then
                wd.log(**missing tag var11**)
                isError = true
        else
                Var11 = wd.getDeviceVar(**TestVirtualDevice**,**var11**)
        end

        local status, message = wd.checkDeviceVar(**TestVirtualDevice**,**var12**)
        if (status == false) then
                wd.log(**missing tag STRING_var12**)
                isError = true
        else
                Var12 = wd.getDeviceVar(**TestVirtualDevice**,**var12**)
        end

        wd.log(**Init OK**)
end

function Sum()
```

```
        Var1.set(Var1.get() + 1)
        Var2.set(Var2.get() + 1)
        Var3.set(Var3.get() + 1)
        Var4.set(Var4.get() + 1)
        Var5.set(Var5.get() + 1)
        Var6.set(Var6.get() + 1)
        Var7.set(Var7.get() + 1)
        Var8.set(Var8.get() + 1)
        Var9.set(Var9.get() + 1)
        Var10.set(Var10.get() + 1)
        Var11.set(Var11.get() + 1)
        Var12.set(**Webdyn**..tostring(Var4.get()))
        wd.log(**Sum=**..Var4.get())
end

local time_sec = 0

function wsTick()
        time_sec = time_sec + 1

        if time_sec >= 60 then
                time_sec = 0
                InverterSum()
        end
end
```

Example log

```
2022-03-18 10:41:19 [test_device_virtuel.lua 14] Start Init
2022-03-18 10:41:20 [test_device_virtuel.lua 20] Init OK
2022-03-18 10:41:21 [test_device_virtuel.lua 57] Sum=1
2022-03-18 10:41:22 [test_device_virtuel.lua 62] Sum=2
Definition file generated by the webdynSunPM:
none;Script;Script;TestVirtualDevice
1;;;STRING;;STRING_var12;STRING_var12;1.000000;0.000000;;4
2;;;U8;;var1;var1;1.000000;0.000000;;4
3;;;F64;;var10;var10;1.000000;0.000000;;4
4;;;F64;;var11;var11;1.000000;0.000000;;4
5;;;U16;;var2;var2;1.000000;0.000000;;4
6;;;U32;;var3;var3;1.000000;0.000000;;4
7;;;U64;;var4;var4;1.000000;0.000000;;4
8;;;I8;;var5;var5;1.000000;0.000000;;4
9;;;I16;;var6;var6;1.000000;0.000000;;4
10;;;I32;;var7;var7;1.000000;0.000000;;4
11;;;I64;;var8;var8;1.000000;0.000000;;4
12;;;F32;;var9;var9;1.000000;0.000000;;4
Sample data file:
SCRIPTINDEX;0
none;WPM0146B6_Script_TestVirtualDevice.csv
12;1;2;3;4;5;6;7;8;9;10;11;12
25/07/31-11:41:00;Test;0;0;0;0;0;0;0;0;0;0;0
25/07/31-11:42:00;Test1;1;1;1;1;1;1;1;1;1;1;1
25/07/31-11:43:00;Test2;2;2;2;2;2;2;2;2;2;2;2
```

4.5.3 updateScriptVars : Updating virtual equipment variables

`<> wd.updateScriptVars (<?> var)`

Update variables to the virtual equipment attached to this script

Parameters

Name	Description
Var	Variables to update in the virtual device

Results

None

Example of creating a virtual device:

```
declarations = {
        sumInverter = 0,
        STRING_messageToDisplay = **No value**
}

local LIMIT = 10000

function wsInit()
        wd.log(**Start Init**)

        wd.declareScriptVars(declarations,60)

wd.log(**Init OK**)
end

function InverterSum()
        val =  GetGeneralPower()
        val = declarations.sumInverter + val
        wd.log(**InverterSum=**..val)
        declarations.sumInverter = val

        if val > LIMIT then
                declarations.STRING_messageToDisplay = **Attention limit exceeded**
                wd.log(**Attention limit exceeded**)
        else
                declarations.STRING_messageToDisplay = **Value ok**
                wd.log(**Value ok**)

        end
        wd.updateScriptVars(declarations)
end

local time_sec = 0

function wsTick()
        time_sec = time_sec + 1

        if time_sec >= 60 then
                time_sec = 0
                InverterSum()
        end
end
```

Example log

```
2022-03-18 10:41:19 [test_device_virtuel.lua 14] Start Init
2022-03-18 10:41:20 [test_device_virtuel.lua 20] Init OK
2022-03-18 10:41:21 [test_device_virtuel.lua 57] InverterSum=2100
2022-03-18 10:41:22 [test_device_virtuel.lua 62] Value ok
2022-03-18 10:42:21 [test_device_virtuel.lua 57] InverterSum=5932
2022-03-18 10:42:22 [test_device_virtuel.lua 62] Value ok
```

```
2022-03-18 10:42:21 [test_device_virtuel.lua 57] InverterSum=10534
2022-03-18 10:42:22 [test_device_virtuel.lua 62] Attention limit exceeded
```

Definition file generated by the webdynSunPM

```
none;Script;Script;test_device_virtuel
1;;;;;messageToDisplay;messageToDisplay;1.000000;0.000000;;4
2;;;;;sumInverter;sumInverter;1.000000;0.000000;;4
```

Sample data file

```
SCRIPTINDEX;0
none;WPM00C73F_Script_test_device_virtuel.csv
2;1;2
22/06/09-09:10:00;Value ok;2100
22/06/09-09:20:00;Value ok;5932
22/06/09-09:30:00;Attention limit exceeded;10534
```

4.5.4 checkVirtualVar : Check equipment variable

`<variable> var wd.checkVirtualVar(<string> tagName )`

Equivalent to :

`<variable> var wd.checkDeviceVar(**<script name>** , <string> tagName )`

The function is used check the script's virtual device has a particular tag This is a helper for having a code independant from script name.

Parameters

Name	Description
TagName	Variable tag name

Results

Name	Description
isError	true: the device exist and the tag is included in its definition file
errorMessage (if isError=true)	Explicit error message (Device not found OR Tag not found)

The variable tag name **TagName** is the name of the **Tag** which must be added with **declareScriptVars** function..

Example

Example of retrieving the value of an equipment variable:

```
function checkVariable()
        isError, errorMessage = wd.checkVirtualVar(**Info**)

        if isError then
                wd.log(**You forget to declare info =**.. errorMessage)
        else
```

```
                wd.log(**variable exists**)
        end
end
```

Example log

```
2022-03-18 14:44:20 [test.lua 107] You forget to declare info =Tag not found
```

4.5.5 getVirtualVar : virtual device variable

`<variable> var wd.getVirtualVar(<string> tagName )`

Equivalent to :

`<variable> var wd.getDeviceVar(**<scriptName>** , <string> tagName )`

The function is used to retrieve the value of script's virtual device variable This is a helper for having a code independant from script name.

Parameters

Name	Description
TagName	Variable tag name

Results

Name	Description
var	A variable corresponding to the parameters:If only one matches the parameters, it will be returned. If no variable matches the parameters, **nil** is returned.

The variable tag name **TagName** is the name of the **Tag** which must be added to the definition files of each device to be taken into account.

Example of retrieving the value of an equipment variable:

```
function GetMyInfo()
        myInfo = wd.getVirtualVar(**Info**)

        if myInfo.get() ~= nil then
                wd.log(**myInfo=**.. myInfo.get())
        else
                wd.log(**no variable declared**)
        end
end
```

Example log:

```
2022-03-18 14:44:20 [test.lua 107] myInfo=test
```

4.6 Plant management functions

The plant management is a high level API which enable scripts to access to plant high level methods without the need to take car of devices and tags.

As described in user manual, devices are organised by category / node Category represents the type of the device : Inverter, Meter ... Node : where the device is connected as its usage, Solar, Grid and later, Storage etc...

Example

- Category Inverter, Node Solar : inverter for PV generation
- Category Meter, Node Solar : Meter measuring the PV generation
- Category Meter, Node Grid : Meter measure the Grid injection/consuption

For each node or device, it's possible to check if a feature is available and if so use the associated methods.

4.6.1 Feature Inverter Basic control

This feature is the possibility for an inverter so support **basic control** i.e. ability to start and stop the PV production of an inverter.

hasBasicControl : check that inverter can be controlled

`<boolean> inverter.hasBasicControl(<string> inverterDeviceName)`

The function is used to check if an inverter can be controlled

Parameters

Name	Description
inverterName	Inverter device name

Results

Name	Description
result	true : inverter exists and it can be controlled, false if not

This functions check the tags CMD_ON Write 1 to Start inverter CMD_OFF Write 1 to Stop inverter

At least one of this tag must exists.

Example checking feature:

```
function checkBasicControl(inverterName)
        if inverter.hasBasicControl(inverterName)
                wd.log(**You can Start/Stop the PV generation for ** .. inverterName)
        else
                wd.log(**Please verify your inverter ** .. inverterName .. ** definition files**)
        end
end
```

Example log:

```
2022-03-18 14:44:20 [test.lua 107] You can Start/Stop the PV generation for INVERTER1
```

setPowerOn : Start stop inverter

`inverter.setPowerOn(<string> inverterDeviceName, <boolean> on)`

The function is used to start or stop an inverter

Parameters

Name	Description
inverterName	Inverter device name
on	true to start inverter, false to stop it

Results

NONE

This function is asynchronous, it means that the control will be executed in background.

This function use the following algorthm

```
inverterSetPowerOn(bool state)
        state = true
                Write 0 to CMD_OFF
                Write 1 to CMD_ON
        state = false
                Write 0 to CMD_ON
                Write 1 to CMD_OFF
```

Example of retrieving the value of an equipment variable:

```
function setInverter1PowerOn(inverterName, state)
        if inverter.hasBasicControl(inverterName)
                inverter.setPowerOn(inverter1, state)
                wd.log(**Command sent to ** .. inverterName)
        else
                wd.log(**Please verify your inverter ** .. inverterName .. ** definition files**)
        end
end
```

Example log:

```
2022-03-18 14:44:20 [test.lua 107] Command sent to INVERTER1
```

4.6.2 Feature Inverter Active control

hasActiveControl : check that active power on an inverter can be controlled

`<boolean> inverter.hasActiveControl(<string> inverterDeviceName)`

The function is used to check if active power of an inverter be controlled

Parameters

Name	Description
inverterName	Inverter device name

Results

Name	Description
result	true : inverter exists and it can be controlled, false if not

This functions check the tags CMD_P_AC_SET_PERCENT : write the power expressed in percent. Unit and scale must be set accordingly. CMD_P_AC_SET_ABS: write the power expressed in W/kW/MW. Unit and scale must be set accordingly.

This functions returns true if one of these tags exists

Example checking feature:

```
function checkActiveControl(inverterName)
        if inverter.hasActiveControl(inverterName)
                wd.log(**You can set the active power of ** .. inverterName)
        else
                wd.log(**Please verify your inverter ** .. inverterName .. ** definition files**)
        end
end
```

Example log:

```
2022-03-18 14:44:20 [test.lua 107] You can set the active power of INVERTER1
```

setRatedPowerKW : Set the rated power of the inveter

`inverter.setRatedPowerKW(<string> inverterDeviceName, <number> ratedPower_kW)`

The function to set the rated power in kW : usefull if you want to set a power in **percent** for an inverter that allows only absolute values.

Parameters

Name	Description
inverterName	Inverter device name
ratedPower_kW	inverter rated power

Set a custom valeur for the rated power of the inverter : no command is sent to inverter. Internal variable containing rated power of inverter is updated.

Results

| Name | Description | NONE

getRatedPowerKW : set the custom rated power value

`<number> inverter.getRatedPowerKW(<string> inverterDeviceName)`

The function returns the rated power of the inverter

Parameters

Name	Description
inverterName	Inverter device name

Results

Name	Description
ratedPower	Rated power

The function returns the rated power :

- if a custom value has been set, return it,
- if a tag P_RATED exists : return the value.
- return 0

Example getRatedPowr:

```
function getRatedPowerKW(inverterName)
        if inverter.hasActiveControl(inverterName)
                wd.log(**Rated power of ** .. inverterName .. ** is ** .. inverter.getRatedPowerKW(i
nverterName) .. ** kW**)
        else
                wd.log(**Please verify your inverter ** .. inverterName .. ** definition files**)
        end
end
```

Example log:

```
2022-03-18 14:44:20 [test.lua 107] Rated power of  INVERTER1 is 100 kW
```

setActivePowerPercent and setActivePowerKW: set the active power

`inverter.setActivePowerPercent(<string> inverterName, <number> percent)`
and `inverter.setActivePowerKW(<string> inverterName, <number> powerKW)`

Parameters

Name	Description
inverterName	Inverter device name
percent	Power in % of the rated power
powerKW	power in kW

Results

NONE

Set the active power value:

```
inverter.setActivePowerPercent(inverter, percent)
        if  CMD_P_AC_SET_PERCENT exists => Write value to this tag
        if inverterGetRatedPowerKW = 0 : error
```

```
         if  CMD_P_AC_SET_ABS :  write (percent * inverterGetRatedPowerKW)

inverter.setActivePowerKW(inverter, power)
         if  CMD_P_AC_SET_ABS :  write (power)
         if inverterGetRatedPowerKW = 0 : error
         if  CMD_P_AC_SET_PERCENT exists => Write power/ratedPower
```

Misc functions to write inverter config

`inverter.setACRampPercentPerSec(<string> inverterDeviceName, <number> rampValue_percent_per_s)` `inverter.setACRevertTimeSec(<string> inverterDeviceName, <number> repeatValue_s)`

Theses functions are used to write configurations in inverters

Parameters

Name	Description
inverterName	Inverter device name
rampValue_percent_per_s	Active power ramp gradient in percent per second
repeatValue_s	setting in inverter used to go back to 100% after a command has been issued

Results

NONE

inverterSetACRampPercentPerSec(inverter, rampValue) if (CMD_P_RAMP_GRADIENT_PPS) write value

inverterSetACRevertTimeSec(inverter, repeatValue) if (CMD_REVERT_TIME_S) write value

4.6.3 Feature Meter Active Power

This feature is the possibility for an meter to provide a measurement of active power.

global record meter hasActivePower: function(meter: string): boolean getActivePowerKW: function(meter: string, min3phase: boolean): number, number end

hasActivePower : check that meter can provide active power

`<boolean> meter.hasActivePower(<string> meterDeviceName)`

The function is used to check if a meter can provide active power measure

Parameters

Name	Description
meterDeviceName	meter device name

Results

Name	Description
result	true : meter exists and it can be controlled, false if not

return true if P_AC tag exists or (P_AC_1 and P_AC_2 and P_AC_3)

getActivePowerKW : retreive active power value

```
<number> delta_value, <number> value getActivePowerKW(<string> meterName, <bool> min3phase)
```

Parameters

Name	Description
meterName	Meter device name
min3phase	true for minimum based computation

min3phase : for zero injection feature, if we want to get sure that we will not inject ON ANY PHASE, then we should reduce the power so the lowest phase is not injecting. It means that the value we should take in account is 3 time the lowest phase

Results

Name	Description
delta_value	age of the measurement expressed in second
value	Active power value in kw

This functions use the following algorthm

```
if min feature :
    if P_AC_1, P_AC_2 and P_AC_3 exists
        return min( P_AC_1, P_AC_2, P_AC_3) *3
    else
        return P_AC
    end
else
    if P_AC exists
        return P_AC
    else
        return P_AC_1 + P_AC_2 + P_AC_3
    end
end
```

4.6.4 Feature Solar Basic control

This feature is the possibility for each solar inverters so support **basic control** i.e. ability to start and stop the PV production of the inverters

hasSolarBasicControl : check that all inverters can be controlled

```
<boolean> plant.hasSolarBasicControl()
```

The function is used to check if all solar inverters can be controlled

Parameters

NONE

Results

Name	Description
result	true : all solar inverters can be controlled, false if not

Check that all solar inverters responds ok to

`<boolean> inverters.hasBasicControl()`

solarSetPowerOn : Start stop inverters

`<boolean> plant.solarSetPowerOn(<boolean> on)`

Parameters

Name	Description
on	true to start solar inverters, false to stop them

Results

NONE

This function is asynchronous, it means that the control will be executed in background.

```
for each inverter in Solar Node
    inverter.setPowerOn(inverter, true)
```

solarHasActiveControl : check if all inverters has active control

`<boolean> plant.solarHasActiveControl()`

Parameters

NONE

Results

NONE

```
for each inverter in Solar Node
    if not inverter.hasActiveControl(inverter) return false

return true
```

solarSetRatedPowerKW : set the active power rated power

`plant.solarSetRatedPowerKW(<number> ratedPowerKW)`

Parameters

Name	Description
ratedPower_kW	solar rated power

Set a custom value for the rated power of the solar : no command is sent to any inverter. Internal variable containing rated power of solar node is updated.

Results

| Name | Description | NONE

solarGetRatedPowerKW : get the rated power value

```
<number> plant.getRatedPowerKW()
```

The function returns the rated power of the solar node

Parameters

NODE

Results

Name	Description
ratedPower	Rated power

The function returns the rated power :

- if a custom value has been set, return it,
- else : get the sum of rated power of all interters in solar node

solarSetActivePowerPercent: set active power in percent

```
plant.solarSetActivePowerPercent(<number> percent)
```

Parameters

Name	Description
percent	active power in percent of rated power

Results

NONE

```
for each inverter in SOLAR node
    inverter.setActivePowerPercent(percent)
```

solarGetRatedPowerKW : get the rated power value

```
<number> plant.getRatedPowerKW()
```

The function returns the rated power of the solar node

Parameters

NODE

Results

Name	Description
ratedPower	Rated power

The function returns the rated power :

- if a custom value has been set, return it,
- else : get the sum of all rated power of interters in solar node

solarSetActivePowerKW: set active power kW

```
plant.solarSetActivePowerKW(<number> power)
```

Parameters

Name	Description
power	active power in kW

Results

NONE

```
percent = power / solarGetActivePower()
for each inverter in SOLAR node
    inverter.setActivePowerPercent(percent)
```

solarGetActivePowerPercent: get active power in percent

```
<number> plant.solarGetActivePowerPercent()
```

Parameters

Results

Name	Description
power	active power in percent

return the last activePower setting in percent or rated power

solarGetActivePowerKW: get active power in kw

```
<number> plant.solarGetActivePowerKW()
```

Parameters

Results

Name	Description
power	active power in kW

return the last activePower setting in kW

solarGetInvertersActivePowerKW: get active power from inverters

```
<number> delta, <number> value plant.solarGetInvertersActivePowerKW()
```

Parameters

Results

Name	Description
delta	Age (in s) of the oldest measure
value	active power in kW read from inverter

return the last activePower in kW read from inverters

Misc functions to write inverters config

```
plant.solarSetACRampPercentPerSec( <number> rampValue_percent_per_s)
plant.solarSetACRevertTimeSec( <number> repeatValue_s)
```

These functions are used to write configurations in inverters

Parameters

Name	Description
rampValue_percent_per_s	Active power ramp gradient in percent per second
repeatValue_s	setting in inverter used to go back to 100% after a command has been issued

Results

NONE

write the settings to each inverters of solar node using:

- inverter.setACRampPercentPerSec(inverter, rampValue)
- inverter.setACRevertTimeSec(inverter, repeatValue)

4.7 MQTT

To use MQTT, WebdynSunPM server 2 must be configured in MQTT mode.

```
<> wd.mqttPublish(<int>serverNumber, <string>topic, <?>payload)
```

Publish MQTT payload

Parameters

Name	Description
ServerNumber	Possible server number: 2. Error if server is not MQTT
Topic	Subject name
Payload	Payload: will convert from lua to json string

Results

None

MQTT post example:

```
function AlarmGeneralPower(power)
        if power == 0 then
                wd.mqttPublish(2,**alarm**,**No energy**)
                wd.log(**No energy**)
        elseif power < 1000 then
                wd.mqttPublish(2,**alarm**,**Low energy:**..power)
                wd.log(**Low energy:**..power)
        end
end
```

Example log

```
2022-03-18 10:41:19 [test.lua 15] Low energy:957
```

4.8 Remote management

The WebdynSunPM integrates Web Services and MQTT allowing to perform remote actions on the scripts of the concentrator.

4.8.1 Web service before version 5

Web services are accessible via HTTP or HTTPS. Web services are based solely on POST requests. If successful, an HTTP 200 code is returned and the data depends on the request. In the event of an error, an HTTP 500 code is returned and the data is:

`#\<ID>/Plain text explaining the error`

Or

`Plain text explaining the error`

Session

A session must be opened before any other commands. This session remains open for 1 minute before it closes automatically if no command is sent.

```
http(s)://<@IP>/login
```

Opening a session in Webservice. A session key cookie will be returned on success. This cookie expires every 1 minute.

Payload input

Name	Description
login_password	JSON file including the concentrator login and password

Cookies In : None **Cookies Out** wdSessionKey : <Random String>

Example of Payload in / JSON:

```
{
        **user**:**userhigh**,
        **password**:**<password/same as website>**
}
```

If the authentication is successful, the return code is HTTP 200 and the payload is a string with the same content as the wdSessionKey cookie: `< Random String >`

Example of wdSessionKey cookie:

`**HCRNCOODQILTBCQE**`

The session key can be used for 1 minute. At each call to a webservice, if the session key changes, a new session key is returned in the session cookie. Thus a query of less than one minute and the systematic use of the returned cookie make it possible to keep the session open. If the session has expired, the following message will be returned: error:Session expired

Script management

```
http(s)://<IP@>/lua?<scriptName>.<functionName>
```

Allows to call a specific function <functionName> of a specific Lua script<scriptName>

Data Input

Params : If necessary, we can send parameters in an object to the function. In JSON format.

Data output

Results : If the function returns an object. In JSON format

Cookies In:wdSessionKey : Previous valid session key

** Cookies Out**: wdSessionKey : New valid session key to use with next request. Can be the one given in request or a new one.

The possible value types for the output data are:

- Absent: the payload will be empty
- Nil: the payload will contain **null**
- A number: the payload will contain the number
- A string: the payload will contain the string between two quotes (JSON string).
- An array/An object: the payload will contain a JSON object or a JSON array

Example lua script test_relay.lua:

```
header = {
        version = 1.1,
        label = **relay control**
```

```
}

local duration_default = 1000

function wsInit()
        wd.log(**Init script**)
        relay = wd.getDeviceVar(**io**,**rel**)
        val = relay.get()
end

function wsStop()
        wd.log(**stop script**)
end

function SetState(parameter)
        local result = {}

        state = parameter.state_relay
        pulse = parameter.pulse
        duration = parameter.duration
        if state ~= nil then
                if state == 0 then
                        OFF()
                else
                        ON()
                end
        end
        if (pulse ~= nil) and (pulse == 1) then
                if duration ~= nil then
                        durationP = duration
                else
                        durationP = duration_default
                end

                Pulse(durationP)
        end

        if relay.get() == 0 then
                result.state_relay = 0
                result.info = **closed**
        else
                result.state_relay = 1
                result.info = **opened**
        end

        return result
end

function ON()
        wd.log(**ON**)
        relay.set(1)
end

function OFF()
        wd.log(**OFF**)
        relay.set(0)
end

function Pulse(durationPulse)
        local val
        wd.log(**Pulse **..durationPulse..**ms**)
        val = relay.get()
        if val == 0 then
                val = 1
        else
```

```
                val = 0
        end
        relay.set(val)
        wd.sleep(durationPulse)
        if val == 0 then
                val = 1
        else
                val = 0
        end
        relay.set(val)
end
```

Remote function call example

Example of POST WebService command:

https://192.168.2.12/lua?test_relay.SetState

Example of data sent in JSON by the web service:

```
{
        **pulse**: 1,
        **length**: 1200
}
```

Example of data received in JSON by the web service:

```
{
        **info**: **closed**,
        **state_relay**: 0
}
```

Examples of Bash to manage the Web Service:

```
#!/bin/bash

echo **====== Example 1 : Using the response as a character string =====**
# **Manual** management of the session cookie to explain how it works.
SESSION_COOKIE=$(curl --silent  -X POST https://$1/login -d '{**user**:**userhigh**,**password**:**h
igh**}')

# remove leading and trailing double quotes
SESSION_COOKIE=${SESSION_COOKIE%**}
SESSION_COOKIE=${SESSION_COOKIE#**}
echo **session cookie=** $SESSION_COOKIE

curl -X POST https://$1/lua?test_relay.SetState  --cookie **wdSessionKey=$SESSION_COOKIE** -d '{**pu
lse**:1,**duration**:300}'
echo


echo **====== Example 2 : Direct use of cookies =====**
# More elegant with curl but hides how session cookie works.
curl -X POST https://$1/login -c session_cookies.txt -d '{**user**:**userhigh**,**password**:**high*
*}'
echo
curl -X POST https://$1/lua?test_relay.SetState  -b session_cookies.txt -d '{**pulse**:
1,**duration**:300}'
echo
```

4.8.2 Web service version 5 or higher

Web services are accessible via HTTP or HTTPS. Web services are based solely on POST requests. If successful, an HTTP 200 code is returned and the data depends on the request. In the event of an error, an HTTP 500 code is returned and the data is:

`#\<ID>/Plain text explaining the error` Or

`Plain text explaining the error`

Session

A session must be opened before any other commands. This session remains open for 1 minute before it closes automatically if no command is sent.

```
http(s)://<@IP>/auth
```

Opening a session in Webservice. A session key cookie will be returned on success. This cookie expires every 1 minute.

Payload input

Name	Description
login_password	JSON file including the concentrator login and password

Cookies In : None **Cookies Out** : wdSessionKey : `\<Random String>`

Example of Payload in / JSON:

```
{
        **user**:**userhigh**,
        **password**:**<password/same as website>**
}
```

If the authentication is successful, the return code is HTTP 200 and the payload is a string with the same content as the wdSessionKey cookie:

< Random String >

Example of wdSessionKey cookie:

`**HCRNCOODQILTBCQE**`

The session key can be used for 1 minute. At each call to a webservice, if the session key changes, a new session key is returned in the session cookie. Thus a query of less than one minute and the systematic use of the returned cookie make it possible to keep the session open.

If the session has expired, the following message will be returned:

`error:Session expired`

Script management

```
http(s)://<IP@>/scripts?<scriptName>.<functionName>
```

Allows to call a specific function _<functionName> of a specific Lua script <scriptName>

Data Input

Params : If necessary, we can send parameters in an object to the function. In JSON format.

Data output

Results : If the function returns an object. In JSON format

Cookies In : wdSessionKey : Previous valid session key

Cookies Out : wdSessionKey : New valid session key to use with next request. Can be the one given in request or a new one.

The possible value types for the output data are:

- Absent: the payload will be empty
- Nil: the payload will contain **null**
- A number: the payload will contain the number
- A string: the payload will contain the string between two quotes (JSON string).
- An array/An object: the payload will contain a JSON object or a JSON array

Example lua script test_relay.lua:

```
header = {
        version = 1.1,
        label = **relay control**
}

local duration_default = 1000

function wsInit()
        wd.log(**Init script**)
        relay = wd.getDeviceVar(**io**,**rel**)
        val = relay.get()
end

function wsStop()
        wd.log(**stop script**)
end

function SetState(parameter)
        local result = {}

        state = parameter.state_relay
        pulse = parameter.pulse
        duration = parameter.duration
        if state ~= nil then
                if state == 0 then
                        OFF()
                else
                        ON()
                end
        end
        if (pulse ~= nil) and (pulse == 1) then
                if duration ~= nil then
                        durationP = duration
                else
                        durationP = duration_default
                end

                Pulse(durationP)
```

```
        end

        if relay.get() == 0 then
                result.state_relay = 0
                result.info = **closed**
        else
                result.state_relay = 1
                result.info = **opened**
        end

        return result
end

function ON()
        wd.log(**ON**)
        relay.set(1)
end

function OFF()
        wd.log(**OFF**)
        relay.set(0)
end

function Pulse(durationPulse)
        local val
        wd.log(**Pulse **..durationPulse..**ms**)
        val = relay.get()
        if val == 0 then
                val = 1
        else
                val = 0
        end
        relay.set(val)
        wd.sleep(durationPulse)
        if val == 0 then
                val = 1
        else
                val = 0
        end
        relay.set(val)
end
```

Remote function call example

Example of POST WebService command:

```
https://192.168.2.12/scripts?test_relay.SetState
```

Example of data sent in JSON by the web service:

```
{
        **pulse**: 1,
        **length**: 1200
}
```

Example of data received in JSON by the web service:

```
{
        **info**: **closed**,
```

```
        **state_relay**: 0
}
```

Examples of Bash to manage the Web Service:

```
#!/bin/bash

echo **====== Example 1 : Using the response as a character string =====**
# **Manual** management of the session cookie to explain how it works.
SESSION_COOKIE=$(curl --silent  -X POST https://$1/auth -d '{**user**:**userhigh**,**password**:**hi
gh**}')

# remove leading and trailing double quotes
SESSION_COOKIE=${SESSION_COOKIE%**}
SESSION_COOKIE=${SESSION_COOKIE#**}
echo **session cookie=** $SESSION_COOKIE

curl -X POST https://$1/scripts?test_relay.SetState  --cookie **wdSessionKey=$SESSION_COOKIE** -d '{
**pulse**:1,**duration**:300}'
echo


echo **====== Example 2 : Direct use of cookies =====**
# More elegant with curl but hides how session cookie works.
curl -X POST https://$1/auth -c session_cookies.txt -d '{**user**:**userhigh**,**password**:**high**
}'
echo
curl -X POST https://$1/scripts?test_relay.SetState  -b session_cookies.txt -d '{**pulse**:
1,**duration**:300}'
echo
```

4.8.3 RPC MQTT

The WebdynSunPM accepts commands received on the **Command** topic and returns the result of the command by publishing on the **Result** topic.

The format of the commands to be placed on the **Command** topic is:

```
{
        **rpcName**:**<scriptName>.<methodName>**,
        **parameters**: **<parameters in JSON>**,
        **callerId**:**<unique string that will be used to identify the message>**
}
```

After executing the command, the WebdynSunPM will publish the result on the **Result** topic:

If the command is executed successfully, we have:

```
{
        **callId**: **<unique string that will be used to identify the message>**,
        **result**: <results in JSON>
}
```

In case of error, we have:

```
{
        **callId**: **<unique string that will be used to identify the message>**,
```

```
        **error**: **<error text>**
}
```

The possible value types for the output data are:

- Absent: the payload will be empty
- Nil: the payload will contain **null**
- A number: the payload will contain the number
- A string: the payload will contain the string between two quotes (JSON string).
- An array/An object: the payload will contain a JSON object or a JSON array

Example lua script test_relay.lua:

```
header = {
        version = 1.1,
        label = **relay control**
}

local duration_default = 1000

function wsInit()
        wd.log(**Init script**)
        relay = wd.getDeviceVar(**io**,**rel**)
        val = relay.get()
end

function wsStop()
        wd.log(**stop script**)
end

function SetState(parameter)
        local result = {}

        state = parameter.state_relay
        pulse = parameter.pulse
        duration = parameter.duration
        if state ~= nil then
                if state == 0 then
                        OFF()
                else
                        ON()
                end
        end
        if (pulse ~= nil) and (pulse == 1) then
                if duration ~= nil then
                        durationP = duration
                else
                        durationP = duration_default
                end

                Pulse(durationP)
        end

        if relay.get() == 0 then
                result.state_relay = 0
                result.info = **closed**
        else
                result.state_relay = 1
                result.info = **opened**
        end

        return result
end
```

```
function ON()
        wd.log(**ON**)
        relay.set(1)
end

function OFF()
        wd.log(**OFF**)
        relay.set(0)
end

function Pulse(durationPulse)
        local val
        wd.log(**Pulse **..durationPulse..**ms**)
        val = relay.get()
        if val == 0 then
                val = 1
        else
                val = 0
        end
        relay.set(val)
        wd.sleep(durationPulse)
        if val == 0 then
                val = 1
        else
                val = 0
        end
        relay.set(val)
end
```

Example of command client publication for Lua script on the Command topic:

```
{
        **rpcName**:**test_relay.SetState**,
        **parameters**: {**pulse**:1,**duration**:400},
        **callerId**: **8c2ed54777684eec8c8c010c16e444df**
}
```

Example of result published by WebdynSunPM on the Result topic:

```
{
        **callerId**:**8c2ed54777684eec8c8c010c16e444df**,
        **result**:{**info**:**closed**,**state_relay**:0}
}
```

4.9 Client-encrypted Lua Script

It is possible to make Lua scripts and encrypt them so that they are not visible to the end customer.

The client-encrypted Lua script file must have the following extension: `.luax`

Below is the script file name format : `\<comment>_.luax`

With:

- `\<comment>` : free field for the user

Example :

`ControlPower.luax`

The operation is identical to a Lua script, except that the script is not modifiable and editable, it is not allowed to export it from the web interface. The use of your encrypted scripts is done in several steps which are:

- Script encryption
- Management of decryption keys in the concentrator.

4.9.1 Client script encryption

Script encryption is based on openssl which uses an implementation of cryptographic algorithms and the SSL/TLS communication protocol. We use AES-128-CBC encryption here. The encryption is done by the **openssl** command and you must provide it with 2 secret keys which are the encryption key and the initial vector. Each secret key must be 16 bytes in hexadecimal.

Example of secret keys:

```
Key=402FC11FA6CE8C4F473196B0B0B01EE4
Initial Vector=F7894FD9EC9D9817F8FB1D6A222B9EDA
```

Example command to encrypt a script with secret keys (command performed under Ubuntu):

```
openssl aes-128-cbc -A -a -e -K 402FC11FA6CE8C4F473196B0B0B01EE4 -iv F7894FD9EC9D9817F8FB1D6A222B9ED
A  -in script.lua -out encrypted_script.luax
```

A new script file will be created which will be encrypted.

4.9.2 Management of decryption keys in the concentrator

The management of decryption keys in the concentrator is something sensitive and is done using specific commands which are:

- **setKey**: Adding keys for decrypting client scripts
- **deleteKey**: Deleting keys for decrypting client scripts

> **Note**
>
> For the operation of the concentrator controls, please refer to the WebdynSunPM manual.

setKey: Adding keys for deciphering client scripts

This command adds the keys to decipher client Lua scripts. The keys must be in a JSON file. The file containing the keys must be available on a server in order to be downloaded by the command.

Example JSON file with keys:

```
{
        **key**:**402FC11FA6CE8C4F473196B0B0B01EE4**,
        **iv**: **F7894FD9EC9D9817F8FB1D6A222B9EDA**
}
```

The command file (FTP, SFTP or WebDAV), MQTT/MQTTS message and SMS is in the following format:

```
setkey=<url>:<interface>:<login>:<password>
```

Parameters:

- url: URL of the file to retrieve. Accepted protocols are HTTP, HTTPS, FTP, SFTP. The port can be specified via the format address:port
- interface: Interface used for the connection: ethernet or modem
- login: server identifier
- password: server password

Return :

- If successful for a JSON command: **OK**.
- If successful for an SMS command: no return.
- If an error is encountered: an explanatory message.

Example command file:

```
[{
    **rpcName**:**sunpm.setKey**,
    **parameters**:{
        **url**:**ftps://ftp.webdyn.com/script_key.json**,
        **interface**:**ethernet**,
        **login**:**login**,
        **password**:**pwd**
    },
    **callerId**:**203**
}]
```

Answer OK:

```
[
    {**callerId**:**203**, **result**:**OK**}
]
```

Or reply with an error:

```
[
    {**callerId**:**203**, **error**:**Invalid interface: ethernet**}
]
```

`deleteKey` : Remove customer ciphering keys

This command removes keys for deciphering client Lua scripts.

The command file (FTP, SFTP or WebDAV), MQTT/MQTTS message and SMS is in the following format:
`deletekey` Return :

- If successful for a JSON command: **OK**.
- If successful for an SMS command: no return.
- If an error is encountered: an explanatory message.

Example command file:

```
[{
    **rpcName**:**sunpm.deleteKey**,
    **callerId**:**117**
}]
```

Answer OK :

```
[
    {**callerId**:**117**, **result**:**OK**}
]
```

4.10 LUA Script with Webdyn license

Lua scripts with Webdyn license are encrypted scripts created by Webdyn. They can be used after purchasing a license.

Such a script file must have the following extension: `.luaw`

Below is the script file name format : `\<scriptwebdyn>_.luaw`

With:

- `\<scriptwebdyn>` : Webdyn proprietary script

Example : `ControlPower.luaw`

The operation is identical to a Lua script, except that the script is not modifiable and editable, it is not allowed to export it from concentrator.

4.10.1 License file

The `\<uid>_licence.ini` file contains the Lua Webdyn `.luaw` script licenses. The license file must be placed in the CONFIG directory of the remote server. The license file makes it possible to obtain specific scripts designed by Webdyn. In this case, please contact the Webdyn sales department who will be able to advise you and redirect you to the contacts concerned: contact@webdyn.com

When connecting to the remote server, if the file is detected, it is downloaded and the license is immediately applied. The WebdynSunPM will not upload the license file if it is deleted.

> **Note**
>
> The license file is specific to a WebdynSunPM, It is not possible to use the same file for several concentrator.

> **Warning**
>
> It's forbidden to modify the contents of the license file under pernalty of blocking the management of concentrator licenses.