WEBDYNSUNPM LUA USER GUIDE

V1.04

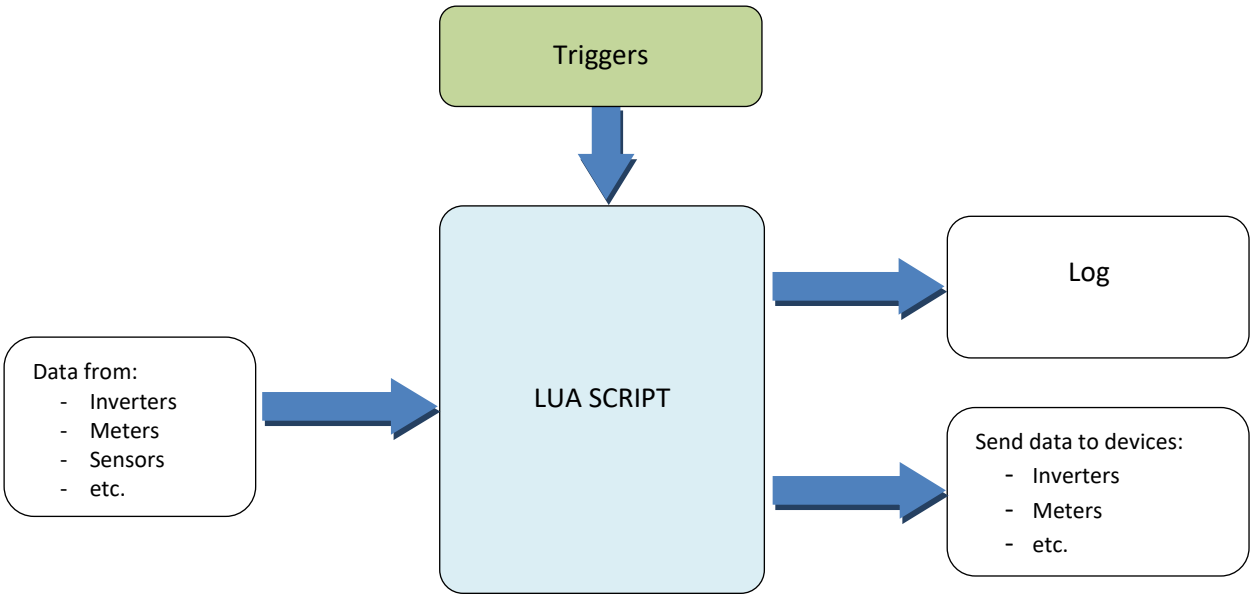
1 The description

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

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 (eg inverters, meters, etc.) and write data to devices.

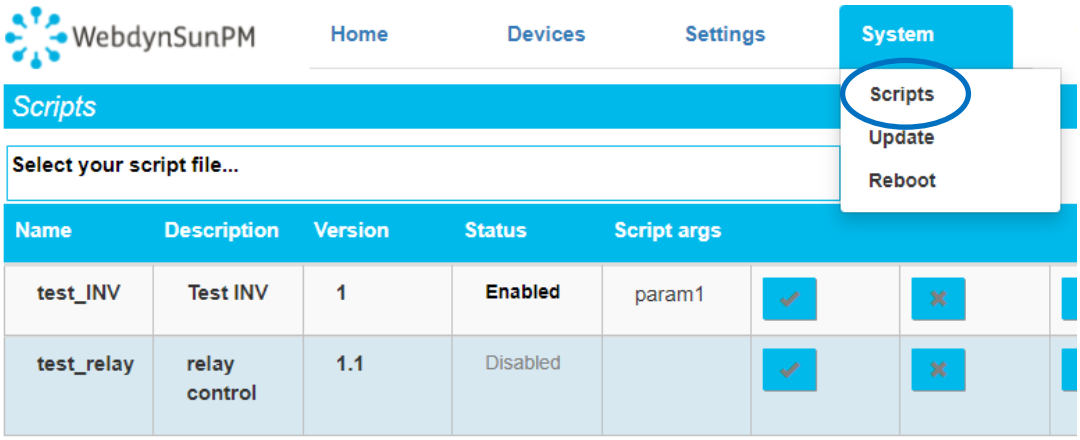
2 *WebdynSunPM configuration*

The configuration of the webdynSunPM for the management of Lua scripts can be done either locally by the integrated Web or by a remote server.

2.1 Import a script

It is possible to import a script by:

- **embedded web interface**:



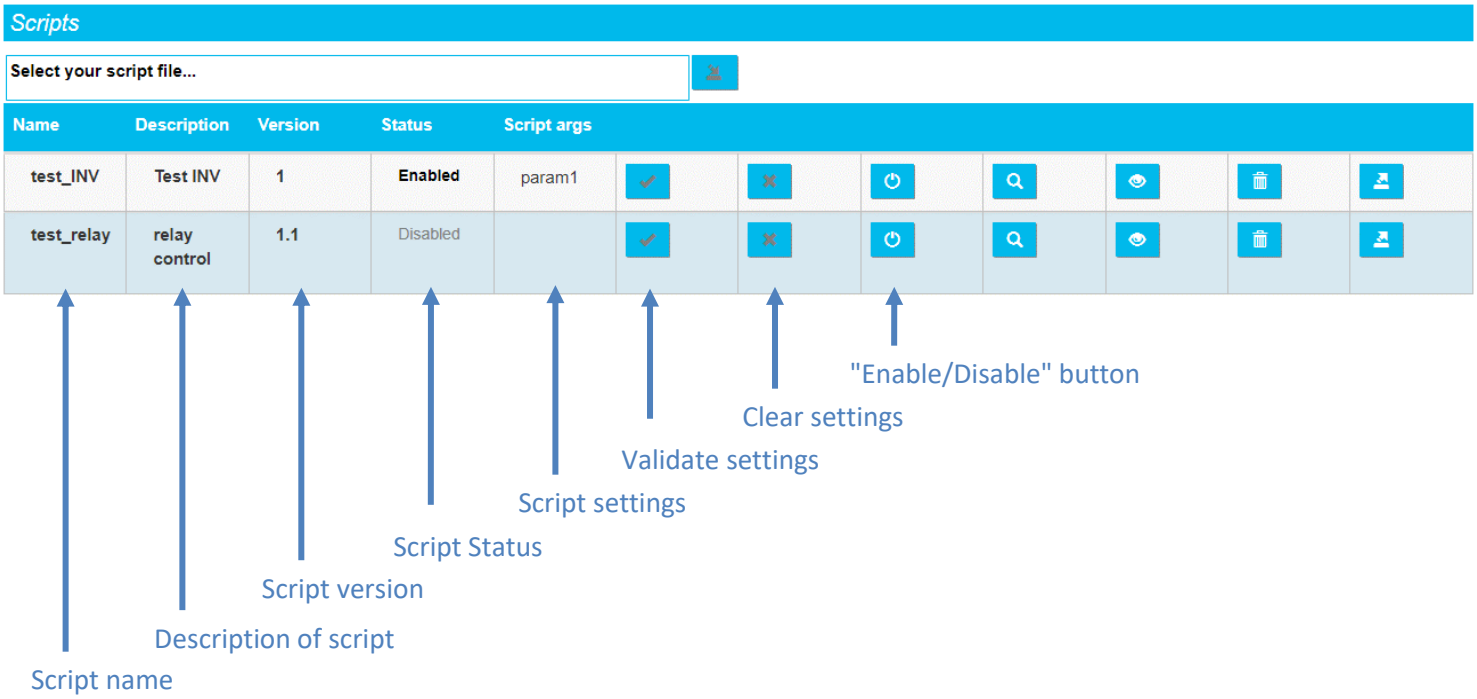
  Go to the "Scripts" page in the "System" tab. Then, click on the “Select your script file…” area, select the Lua file to import and click on the import logo.
  Upon successful import, the Lua script will be displayed in the table below. (see in the WebdynSunPM manual chapter 3.2.3.1.1: “Importing a script”)

- **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 chapter 3.1.2.3.4: “File “<uid>_scl.ini”)

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 .

2.2 State of a script

When a script is imported through the embedded web interface, it is disabled by default. To change the status of a script, click on the “Enable/Disable” button and check its displayed status.



The name of the script displayed corresponds to the name of the file without its lua extension.

A script can have 3 states which are:

- **Enabled**: The script is loaded, the syntax has been validated, each feature coded in the script is activated. The settings of the script are taken into account during its activation.
- **Disabled**: the script is loaded; the syntax has been validated but no functionality is enabled. This script can be activated at the request of the user.
- **Error**: An error has been detected, the script is disabled: no functionality is enabled. User modification of the script is required.

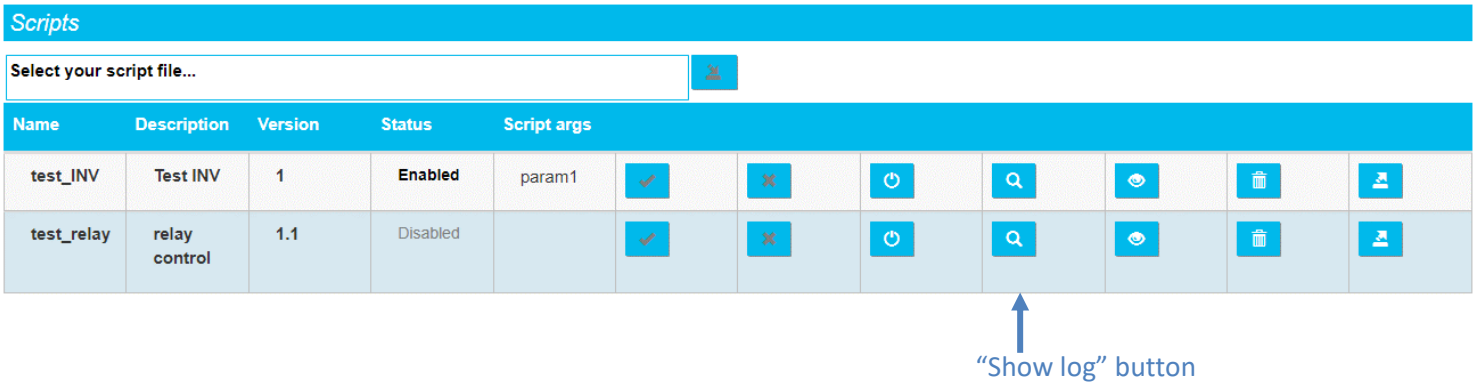
When a script is imported by a remote server, the script is in the state indicated in the "SCRIPT_Enable" variable of the “<uid>_scl.ini” file.
When a script is imported through the embedded web interface, the script is disabled by default.

When a script is stopped, its context is removed. All used Lua variables are cleared.

When the webdynSunPM is restarted, all scripts return to the same state as before. For example, if the script was started then it will be started.

2.3 Script Log

The logs of each script are visible directly from the embedded web interface, by clicking on the "Show Log" button.



At each connection, the logs of each script are sent to the remote server. All the script logs are in the remote "LOG" directory. In this directory, there is one log file per script and per connection. The name of the file is named as follows:

<uid>_LUA_<name_script>_<timestamp>.log.gz

With :

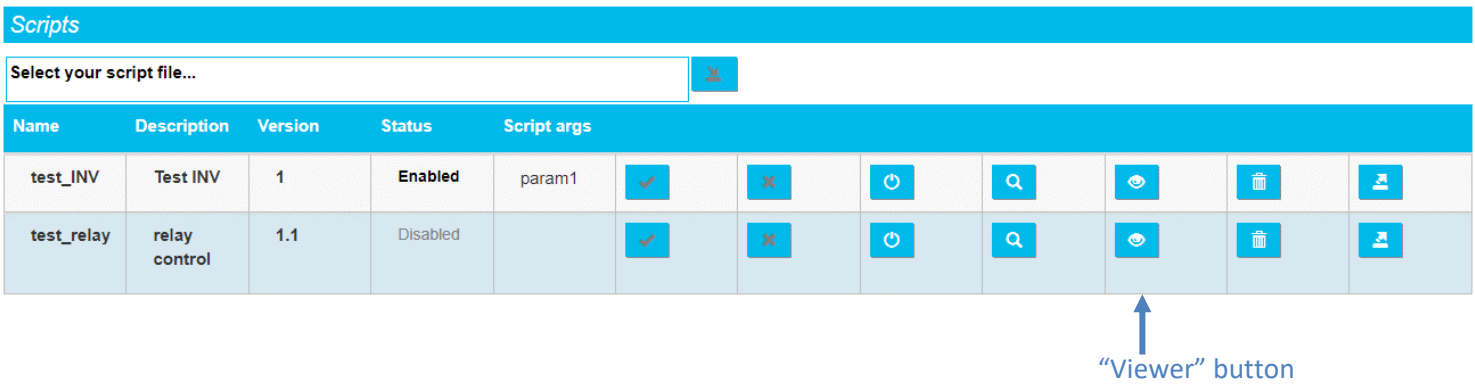
- *<uid>*: Concentrator identifier
- *<name_script>* : script name
- <timestamp>: The timestamp format is "YYMMDD_HHMMSS" so that an alphabetical sorting of the directory gives the chronological order

Examples:

WPM00C73F_LUA_test_INV_220314_091336.log.gz
WPM00C73F_LUA_test_relay_220314_091336.log.gz

2.4 View the contents of a script

Once a script has been imported into the WebdynSunPM, it is possible to see its content on the embedded web interface by clicking on the "Viewer" button.

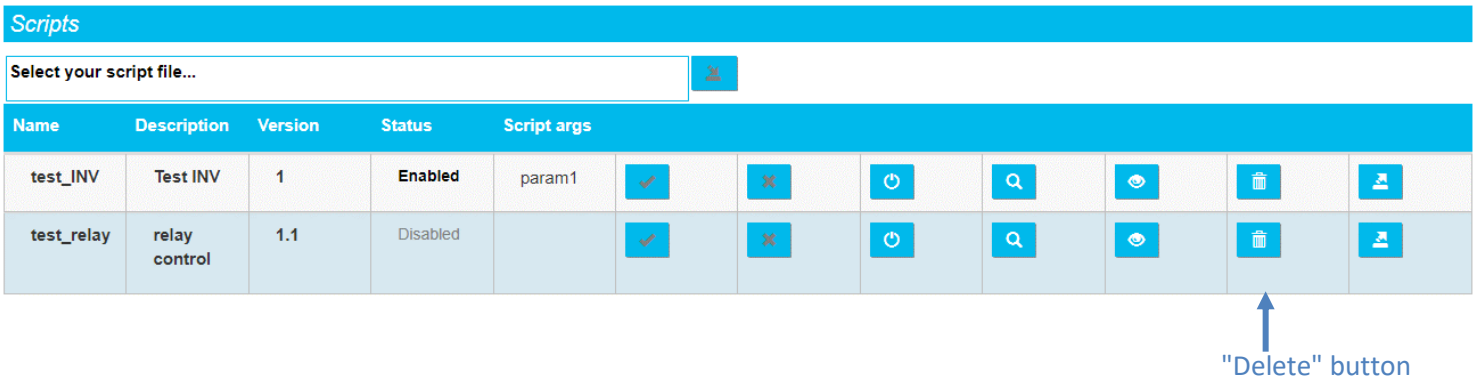


All the scripts remain available on the remote server in the "SCRIPT" directory, even those imported locally.

> Webdyn proprietary ".luaw" scripts and encrypted client ".luax" scripts are not visible and the button is not available.

2.5 Delete a script

Deletion can be done on the embedded web interface, by clicking on the "Delete" button.



You can also delete a script by editing the script configuration file present in the remote “CONFIG” directory. This file is called:

<uid>_scl.ini

With :

- *<uid>*: Concentrator identifier

Example :

WPM00C73F_scl.ini

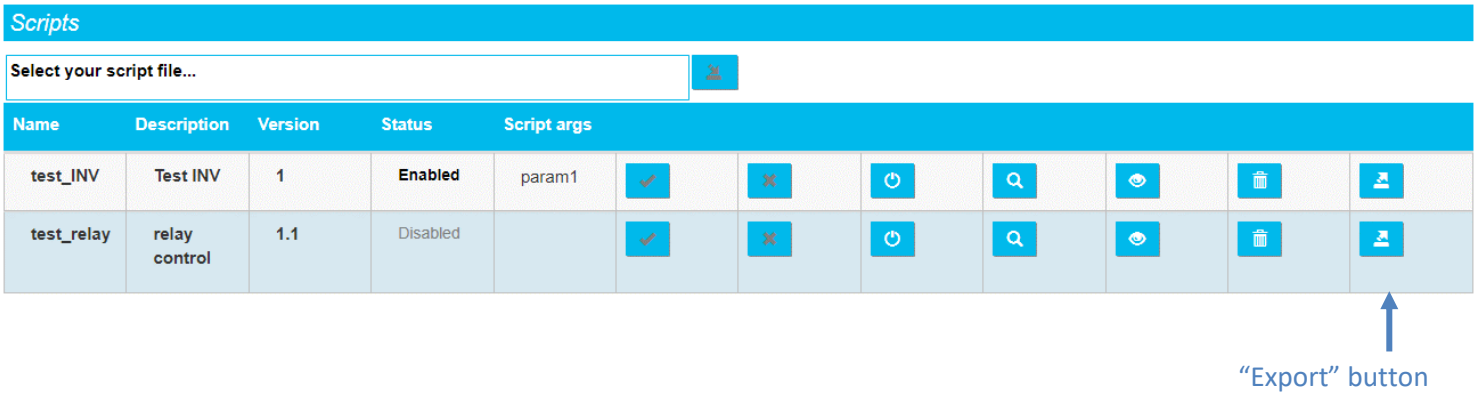
The file includes an index per script, just delete the lines corresponding to the index of the script you want to delete. Care must be taken to adjust the index of the other scripts present, ensuring that the index starts at 0 and that it follows each other. It is imperative that the index be unique.

Example of script configuration file:

```
SCRIPT_File[0]=test_INV.lua
SCRIPT_Enable[0]=1
SCRIPT_Args[0]=param1
SCRIPT_File[1]=test_relay.lua
SCRIPT_Enable[1]=0
SCRIPT_Args[1]=
```

2.6 Export script

From the embedded web interface, you can export a script. To do this, click on the "Export" button.



Webdyn proprietary ".luaw" scripts and encrypted client ".luax" scripts are not exportable and the button is not available.

3 Lua file

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 = {
        release=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 ()
        return1
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

3.1 « wsInit » : Script initialization

[<string> error] **wsInit**([<string> Parameters])		
Function called by the WebdynSunPM when the script is activated		
Parameters		
	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		
	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
```

3.2 « 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
```

3.3 « wsStop » : Stopping the script

<> **wsStop**()	
Function called by the WebdynSunPM when the script is going to be stopped	
Settings	
	Any
Results	

	Any

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:

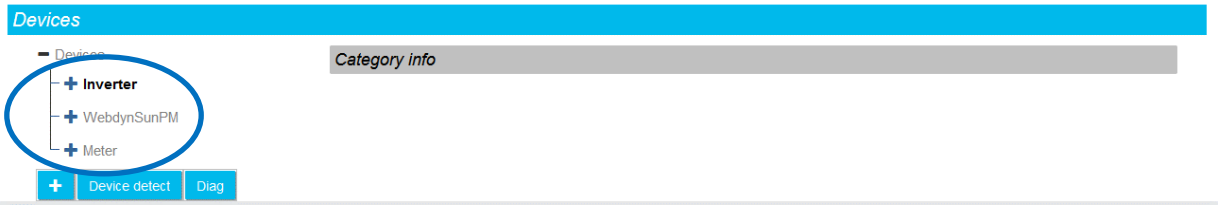
```
2022-03-17 14:47:36 [test.lua 55] script stopped
```

3.4 WebdynSunPM functions

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.

3.4.1 Equipment category

Categories identify a group of equipment 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 “Devices” tree:



In the equipment definition file "<uid>_daq.csv", the category is part of the parameters of each equipment. (See the WebdynSunPM manual chapter 3.1.2.1.3.4: "Declaration of equipment to be supervised".)

3.4.1.1 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		
	Listing	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
```

3.4.1.2 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		
	Category name	Device name as it appears in web pages.
	tagName	Variable tag name
Results		
	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. (see the WebdynSunPM manual chapter 3.1.2.2.2: “Content of the definition file”)

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
```

3.4.1.3 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		
	Category name	Category name
	tagName	Variable tag name to monitor
	Recall	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. (see the WebdynSunPM manual chapter 3.1.2.2.2: “Content of the definition file”)

The "wd.onCategoryVarChange" function systematically triggers the callback function at startup even before the 1st change.

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()
```

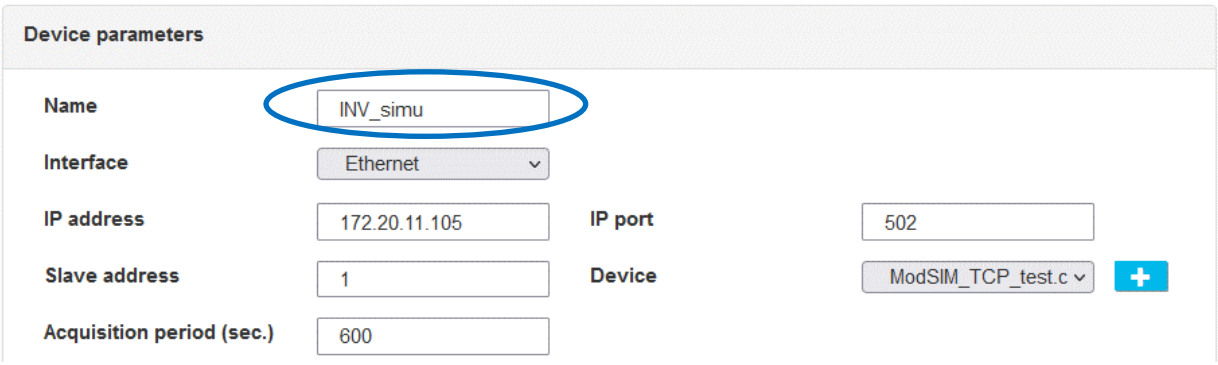
```
    local PacTotal = 0

    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) then
                     wd.log("INV["..i.."]="..PoweracVars[i].get())
                     PacTotal = PacTotal + PoweracVars[i].get()
                else
                     wd.log("INV["..i.."]=null")
          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 10:43:37 [test.lua 88] PacTotal(3 Inverters)=10063W
```

3.4.2 Equipment

A device is a device connected to the WebdynSunPM. The inputs/outputs of the WebdynSunPM are also considered as equipment.
A device has a unique name that identifies it.
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. (see the WebdynSunPM manual chapter 3.1.2.3.4: “Declaration of equipment to be supervised”)

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. (see the WebdynSunPM manual chapter 3.1.2.2.2: “Content of the definition file”)

3.4.2.1 Equipment status

<variable> state **wd.getDeviceStatus**(<string> deviceName)		
The function allows the state of a device		
Parameters		
	deviceName	Equipment name
Results		
	State	Equipement status : • **0**: 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 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
```

3.4.2.2 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		
	Equipment name	Equipment name

	TagName	Variable tag name
Results		
	isError	true: the device exist and the tag is included in its definition file false:
	errorMessage	If isError = true : Explicit error message (Device not found OR Tag not found) Else Empty string

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. (see the WebdynSunPM manual chapter 3.1.2.2.2: “Content of the definition file”)

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
```

3.4.2.3 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		
	Equipment name	Equipment name
	TagName	Variable tag name
Results		
	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. (see the WebdynSunPM manual chapter 3.1.2.2.2: “Content of the definition file”)

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
```

3.4.2.4 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		
	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. (see the WebdynSunPM manual chapter 3.1.2.2.2: “Content of the definition file”)

The “wd.onDeviceVarChange” function systematically triggers the callback function at startup even before the 1st 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
```

3.4.3 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.

3.4.3.1 Reading an object variable

<?> value **variable.get**()		
Function that returns the last known value of a variable.		
Parameters		
	None	
Results		
	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
```

3.4.3.2 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		
	Value	Value of the variable. The type depends on the type of value

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
```

3.4.3.3 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	Information structure : • **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
```

3.4.3.4 Writing an object variable

<Boolean> result , [<string> message] **variable.set**(<?> value)
Function that allows you to modify the value of a variable.
Parameters

	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			
	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
```

3.4.4 Systems

3.4.4.1 Log

<> **wd .log**(<?> message)		
Function called by script log a line.		
Parameters		
	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
```

3.4.4.2 Save a configuration

<> **wd.save**(<?> config)		
Save an object as script parameters		
Parameters		
	Configuration	Configuration to save as script settings
Results		
	None	

The configuration must be of object type only.

Deleting the script file also deletes the settings backup attached to it.

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
```

3.4.4.3 Loading a configuration

<config> **wd.load**()		
Loading script parameters into an object		
Parameters		
	None	
Results		
	Configuration	Configuration saved as script settings

The configuration must be of object type only.

Deleting the script file also deletes the settings backup attached to it.

Example of loading 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 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
```

3.4.4.4 Delay timer

<> **wd.sleep**(<> time_ms)		
Wait time in milliseconds before the next instruction		
Parameters		
	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
```

3.4.4.5 Send alarms

<> wd.sendAlarm(<string> alarmType, <string> msg, <Boolean> delayed)		
Send an alarm to the server		
Parameters		
	alarmType	Alarm type in the alarm file
	msg	Message for the alarm
	Delayed	Can be omitted : defaut value is false

		True: the alarm it delayed (will be sent at the next connection)
Results		
	None	

Example of a 1500ms delay:

```
function triggerAlarm()
    wd.sendAlarm("myType", "seriousAlarm")
end
```

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

```
23/01/10-12:04:05;myType;seriousAlarm
```

3.4.5 Virtual equipment

From a Lua script, it is possible to create a virtual device. A script can declare variables on its virtual equipment which can be written and read, the values of the variables will be saved in the DATA directory on the remote server like the data of an actual equipment. Of course, this virtual device has no physical existence and is never collected. The virtual equipment is only visible by Lua scripts and a corresponding definition file will be deposited by the WebdynSunPM in the DEF directory (equipment definition) on the remote server. This file is for information purpose only, no changes to the virtual device definition file are accepted.

Below is the data file name format:

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

With :

- <uid> : Concentrator identifier
- <script_name> : Script file name
- <script_version>: Script version

Content of the definition file :

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	Value : « none »
Category	Value : « Script »
Manufacturer	Value : « Script »
Model	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:

Champ	Description
Index	Contains the unique identifier of the variable in the file.
Info1	None
Info2	None
Info3	Variable format. The allowed formats are: •F64: floating on 64 bits (8 bytes, or 4 registers) •String: the variable is a character string
Info4	None
Name	Variable name as declared in the script.
Tag	Variable tag as declared in the script.
CoefA	Ignored
CoefB	Ignored
Unit	Ignored
Action	The possible action is only of type: • 4: the variable is of the instant value type.

Note that variables are not private for a Lua script. Thus, another script can read or write the variables of another virtual device. But only a script can declare the variables in its associated virtual device.

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
```

Colour code:

- Black: fixed text.
- Blue: device-specific information or data.

With:

- NumScript_N: index of the script in the script configuration file « <uid>_scl.ini » (see the WebdynSunPM manual chapter 3.1.2.1.4 “File “<uid>_scl.ini””)
- fileDefinitionName_N: definition file name for virtual device N

The data has the same format as the other equipment. (see the WebdynSunPM manual chapter 4.1.3.3 “Data”)

<> **wd.declareScriptVars** (<?> var, <> AcqPeriod)		
Declare variables to the virtual equipment attached to this script		
Parameters		
	var	Variables to attach to the virtual device
	AcqPeriod	The acquisition period in seconds. No parameter, default 600 seconds = 10 minutes

Results		
	None	

<> **wd.updateScriptVars** (<?> var)		
Update variables to the virtual equipment attached to this script		
Parameters		
	Var	Variables to update in the virtual device
Results		
	None	

Example of creating a virtual device:

```
declarations = {
        sumInverter = 0,
        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.messageToDisplay = "Attention limit exceeded"
           wd.log("Attention limit exceeded")
     else
           declarations.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
```

3.4.6 MQTT

To use MQTT, WebdynSunPM server 2 must be configured in MQTT mode. (see the WebdynSunPM manual chapter 3.2.2.5.4 “MQTT”)

<> **wd.mqttPublish**(<int>serverNumber, <string>topic, <?>payload)		
Publish MQTT payload		
Parameters		
	ServerNumber	Possible server number: • 2: WebsunPM server 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 Remote management

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

4.1 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
```

4.1.1 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		
	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
```

4.1.2 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
```

Example of POST WebService command:

```
http://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 http://$1/login -d '{"user":"userhigh","password":"high"}')

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

curl -X POST http://$1/lua?test_relay.SetState  --cookie "wdSessionKey=$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 http://$1/login -c session_cookies.txt -d '{"user":"userhigh","password":"high"}'
echo
curl -X POST http://$1/lua?test_relay.SetState  -b session_cookies.txt -d '{"pulse":1,"duration":300}'
echo
```

4.2 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
```

4.2.1 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		
	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
```

4.2.2 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
```

Example of POST WebService command:

```
http://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 http://$1/login -d '{"user":"userhigh","password":"high"}')

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

curl -X POST http://$1/lua?test_relay.SetState  --cookie "wdSessionKey=$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 http://$1/login -c session_cookies.txt -d '{"user":"userhigh","password":"high"}'
echo
curl -X POST http://$1/lua?test_relay.SetState  -b session_cookies.txt -d '{"pulse":1,"duration":300}'
echo
```

4.3 RPC MQTT

The WebdynSunPM accepts commands received on the "Command" topic and returns the result of the command by publishing on the "Result" topic. (see the WebdynSunPM manual chapter 3.2.2.5.4 “MQTT”)

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

```
{
    "rpcName":"<scritName>.<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}
}
```

5 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.

5.1 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
F7894FD9EC9D9817F8FB1D6A222B9EDA  -in script.lua -out encrypted_script.luax
```

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

In order to guarantee the interest of the function, the unencrypted source file and the secret keys should never be shared.

5.2 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

For the operation of the concentrator controls, please refer to the chapter 5 : “Commands” in the WebdynSunPM manual.

5.2.1 “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":"ftp://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"}
]
```

5.2.2 « deleteKey » : Remove customer ciphering keys

This command removes keys for deciphering client Lua scripts.

If scripts in ".luax" format are present after the decryption keys are deleted, they will continue to work as long as the script remains active and the hub is not restarted. It is strongly recommended to delete ".luax" scripts after deleting the keys.

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"}
]
```

6 *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.

6.1 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.

The license file is specific to a WebdynSunPM. It is not possible to use the same file on several concentrators.
It is forbidden to modify the content of the license file, under penalty of blocking the management of the concentrator licenses.

7 Examples

Specific Lua application notes are available for download on our website :
https://www.webdyn.com/support/