www.AKCP.com

SP+ Scripting Manual

A new feature has been added to sensorProbe+ family: Lua Script Sensor, custom scripting for sensor value calculations.
These scripts support using the Lua programming language's math library.

What is Lua?

Lua is a powerful, efficient, lightweight, embeddable scripting language. It supports procedural programming, object-oriented programming, functional programming, data-driven programming, and data description.
Lua combines simple procedural syntax with powerful data description constructs based on associative arrays and extensible semantics. Lua is dynamically typed, runs by interpreting bytecode with a register-based virtual machine, and has automatic memory management with incremental garbage collection, making it ideal for configuration, scripting, and rapid prototyping.

SP+ implementation

The following Lua 5.3 language features are fully supported: variables, tables, loops, conditions, functions.
The limitations for running scripts on SP+ is up to 4KB script size and 30KB runtime memory; up to a maximum of 80 individual scripts can be defined.

As noted earlier, SP+ only supports the math library of Lua. For example, the following is valid:

```
return math.sqrt(math.pi)
```

For tutorials about how to use the Lua math library, please refer to these pages:
http://lua-users.org/wiki/MathLibraryTutorial
https://www.tutorialspoint.com/lua/lua_math_library.htm
https://www.lua.org/manual/5.3/manual.html#6.7

SP+ provides 5 main Lua libraries for the custom scripts. Mostly the Sensor and Math libraries are used by custom script calculations.

1. Sensor: The Sensor library provides functions to access the sensors connected to the SP+ device. The "vsens" library can be used to access virtual sensors.
2. Math: The Math library comprises a standard set of mathematical functions.
3. Operating system: This library includes functions for getting the current system time.
4. Script: This library provides information about the currently running Lua script.
5. Storage: This library allows to keep values between Lua interpreter runs.

Please refer to the "*List of all available commands*" section at the end of this manual for the list of all functions list for these libraries.

You can access other sensors' values by calling the `sensor.find()` function.
The function has one argument: sensorId, and returns a table with fields:
`id, status_update_time, value, status, acknowledge`

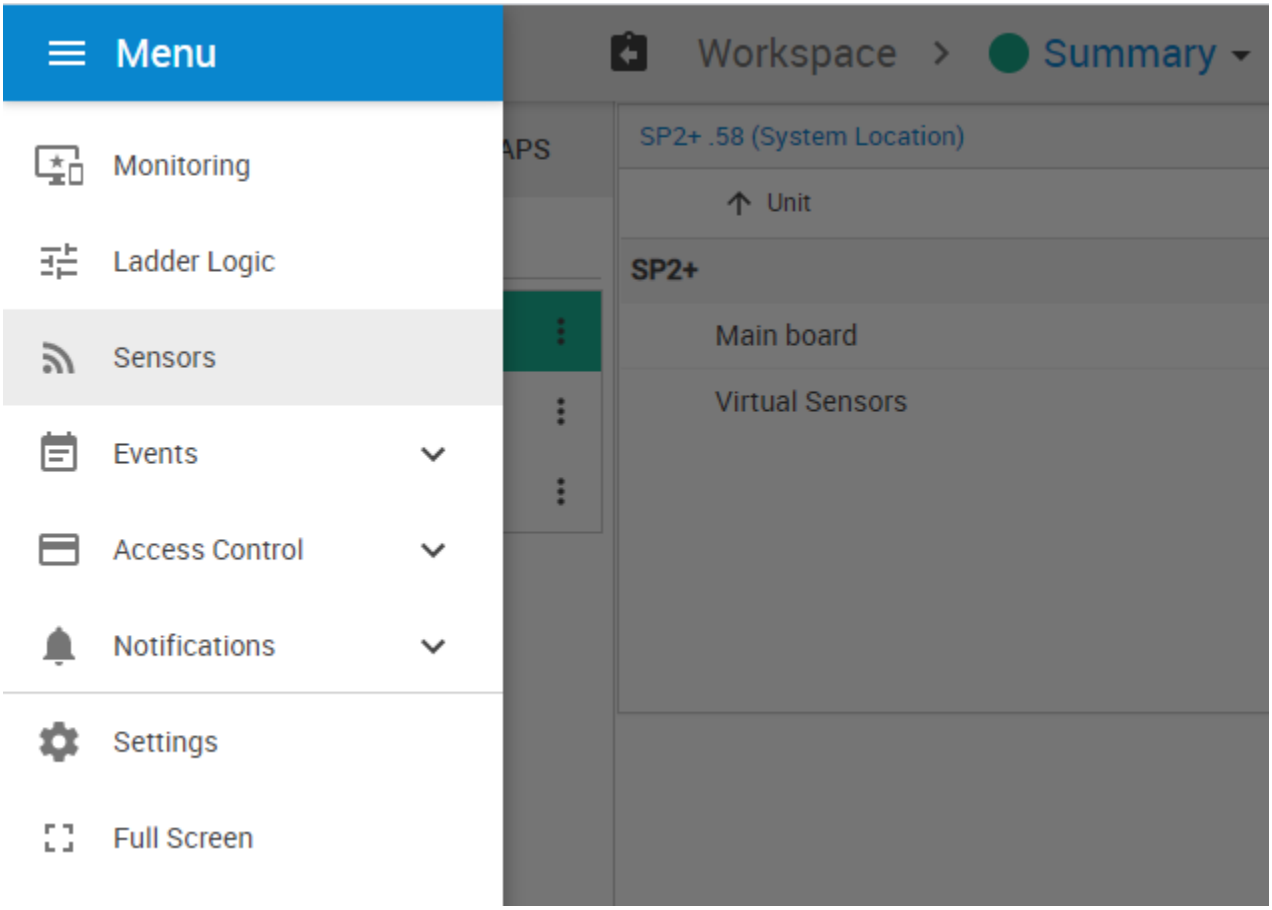
For example, using the physical RJ45 Sensor Port 1's value:

```
s = sensor.find(0)
return s.value * 5
```

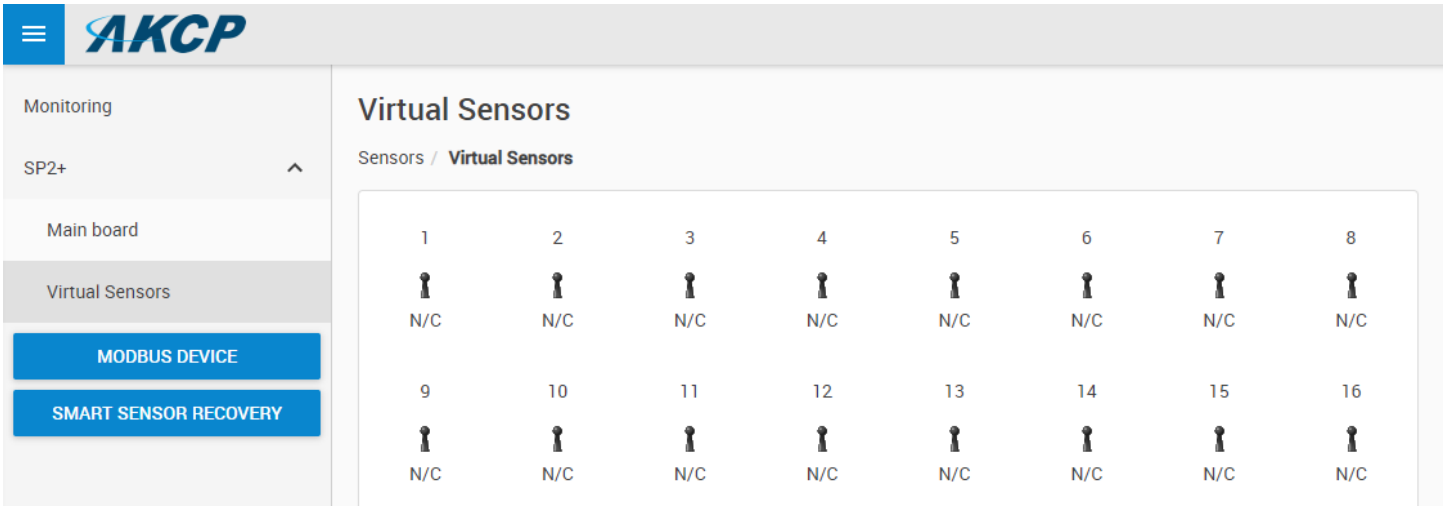
To find the correct Sensor ID, you should use the "Add Command" wizard (see below in the manual).

Script sensor usage

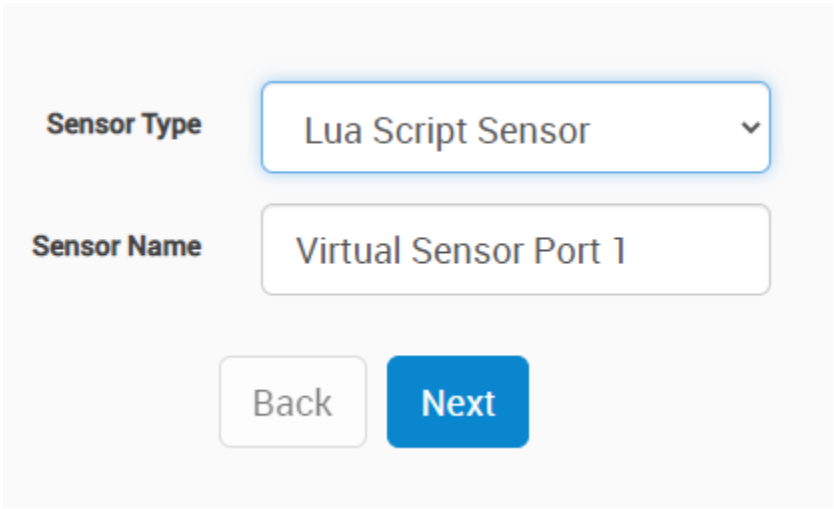
You can add a new script sensor the usual way through the Virtual Sensors interface.
Open the **Menu / Sensors**:



Choose **Virtual Sensors** (note: you may need additional licenses) and click on a new VS:

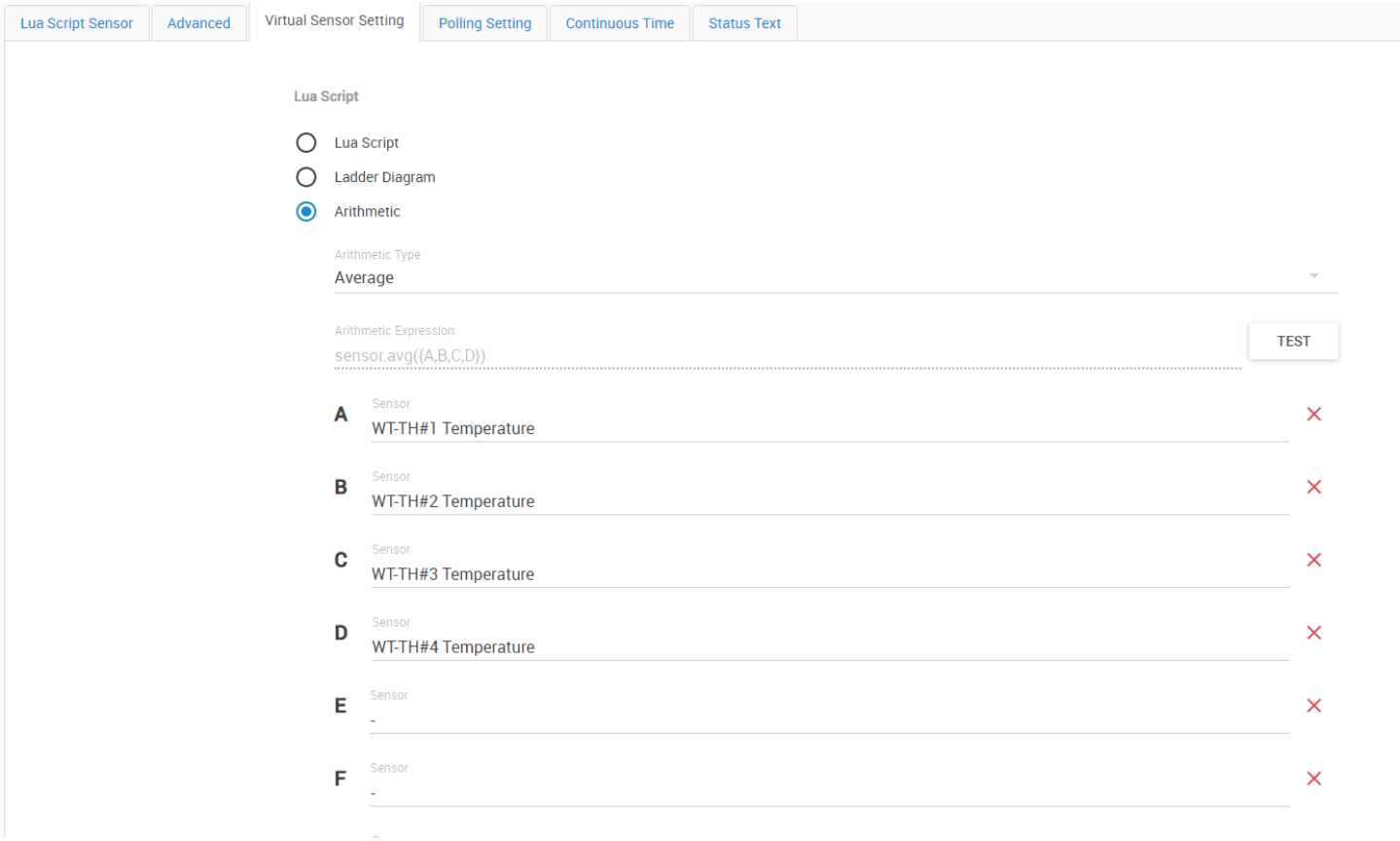


Choose **Lua Script Sensor** from the drop-down menu and type in a custom **Sensor Name** (up to 16 characters):



When you are creating a new Lua Script Sensor, the script ID will be automatically assigned to it, depending on the chosen VS number.
For example, choosing VS #1 will give your script sensor the ID #1. Choosing VS #5 will have ID #5. Up to 80 IDs can be used (id:1..80).

If you would only like to make simple arithmetic calculations on sensor values, you may choose **Arithmetic** type and use the sensor selection drop-down menus then choose your calculation method:

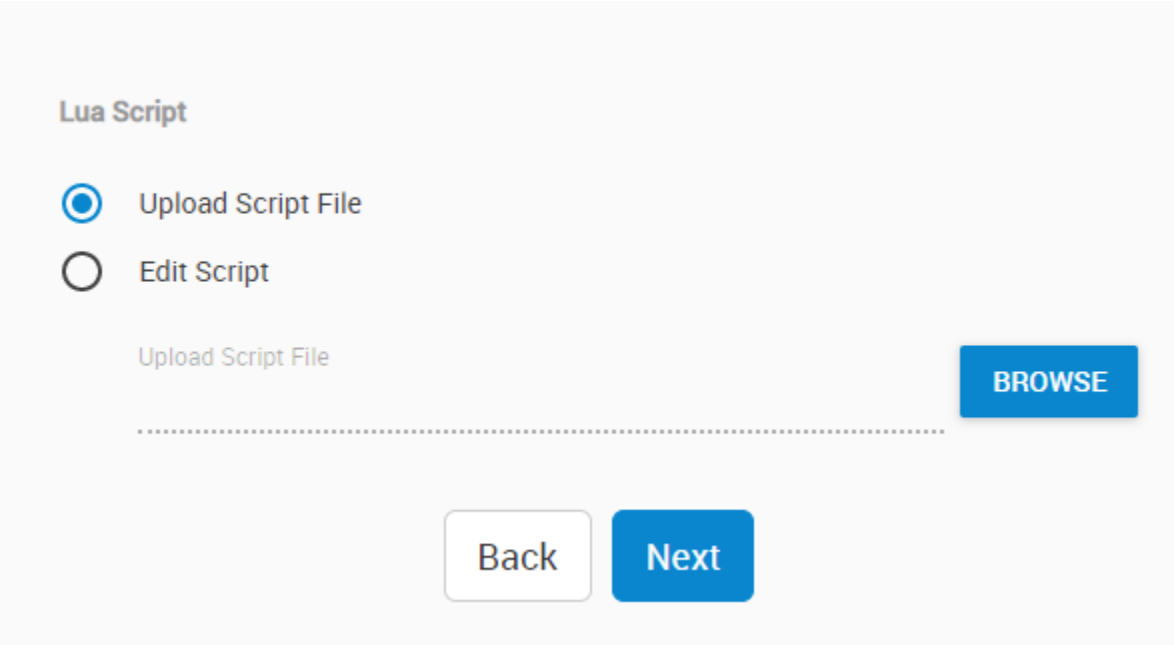


Click on the **Test** button to see a result of the calculation, which will also be the new sensor's value. In this manual we focus on the custom scripting; using the arithmetic calculations are straight-forward.

Note: sensors with invalid status will not be used in the value and status calculation of the Lua Arithmetic Virtual Sensor.

For a Lua custom script sensor, you will need to define the script's contents.
You can either add a script by uploading a predefined script file, or by editing the script directly in the WebUI.
To remove your script, offline the virtual sensor.

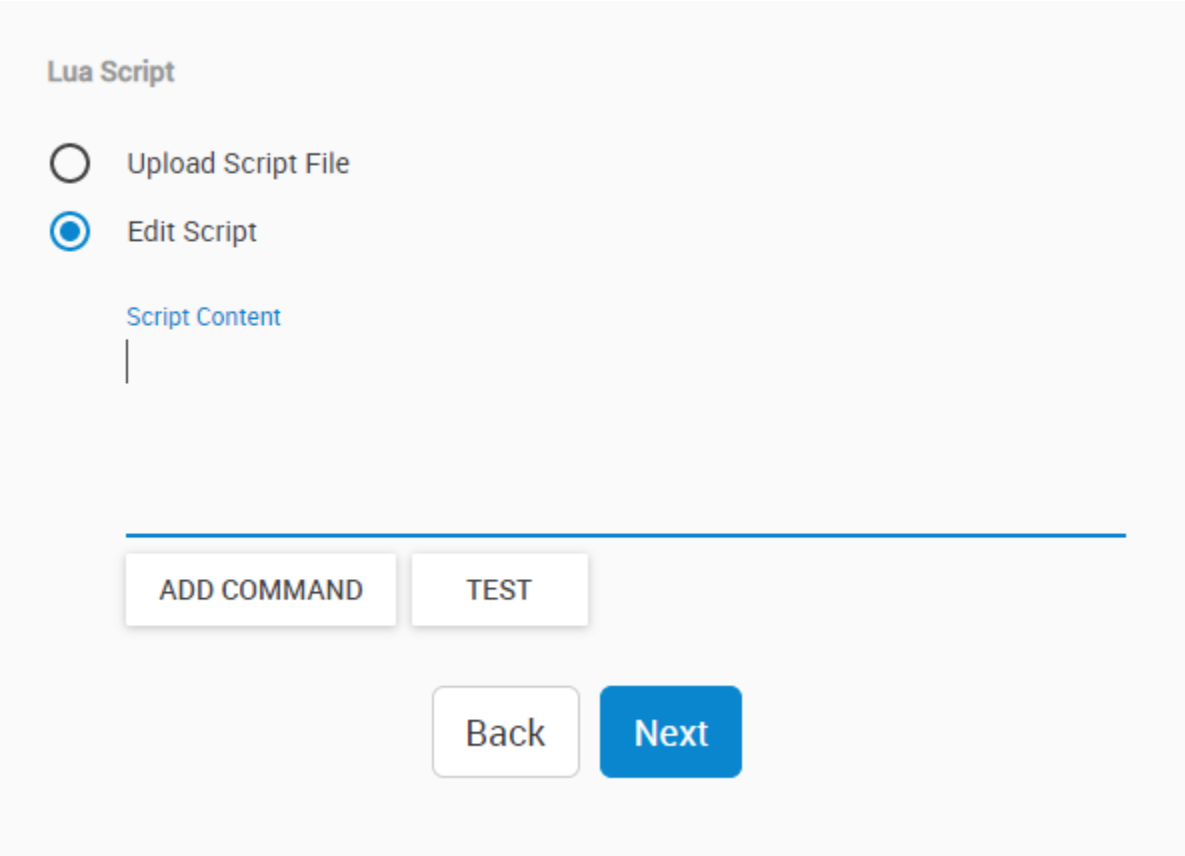
To upload a script, click on **Browse** and select it from your computer:



Notes:

- You can upload files up to 4KB size
- The file extension must be .LUA
- The line ending format could be either Windows (CR LF) or Unix (LF), both are supported

To edit the script in WebUI, you can either type in the script directly, or use the built-in helper functions by using the **Add Command** button (see details below):



The executed Lua script will return a value (integer, float or boolean types).
The returned value will be used as virtual sensor value.
For example, the simplest working Lua script is:

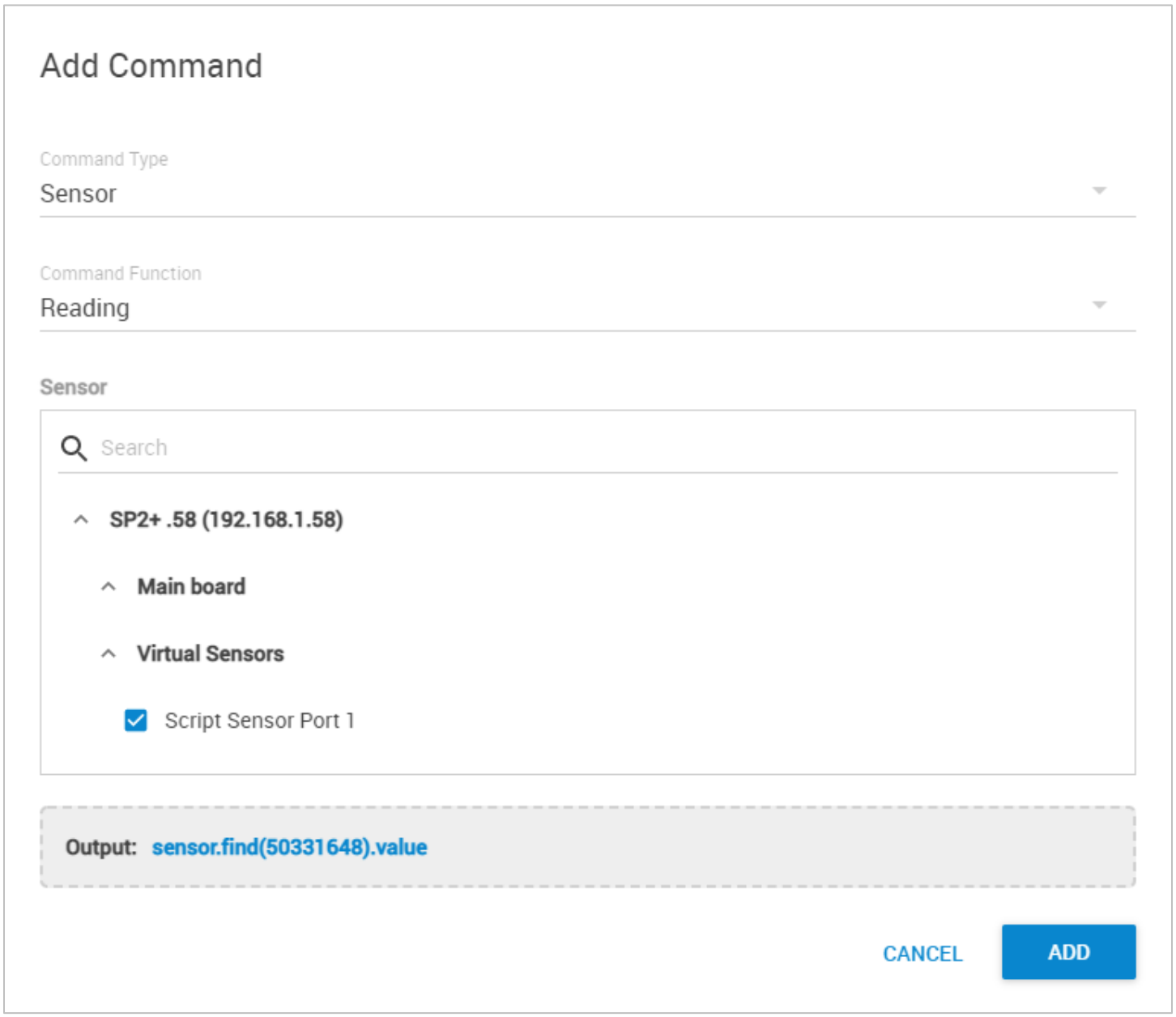
```
return 45
```

Which will return the static integer value 45.

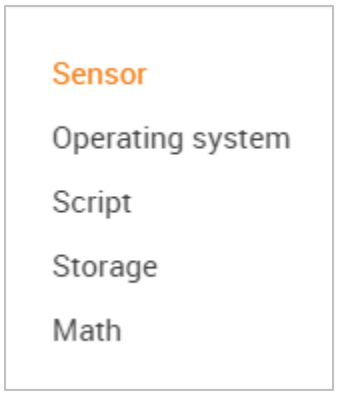
Using the **Add Command** menu will display a wide range of selectable commands.
Please refer to the "*List of all available commands*" section at the end of this manual for the list of all available choices and combinations.

For example, choose a Sensor **Command Type** (default selected).
Then as **Command Function** the sensor's reading.
Then choose the actual sensor on your device (physical or virtual) from the sensors list.

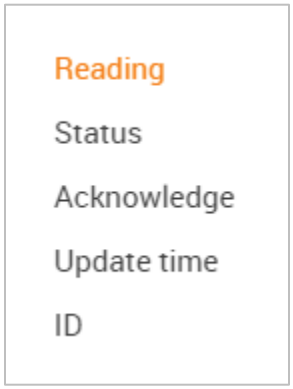
The "**Output**" line will show the actual correct command that the wizard will add to your script file:



There are many other kinds of **Command Types** available:

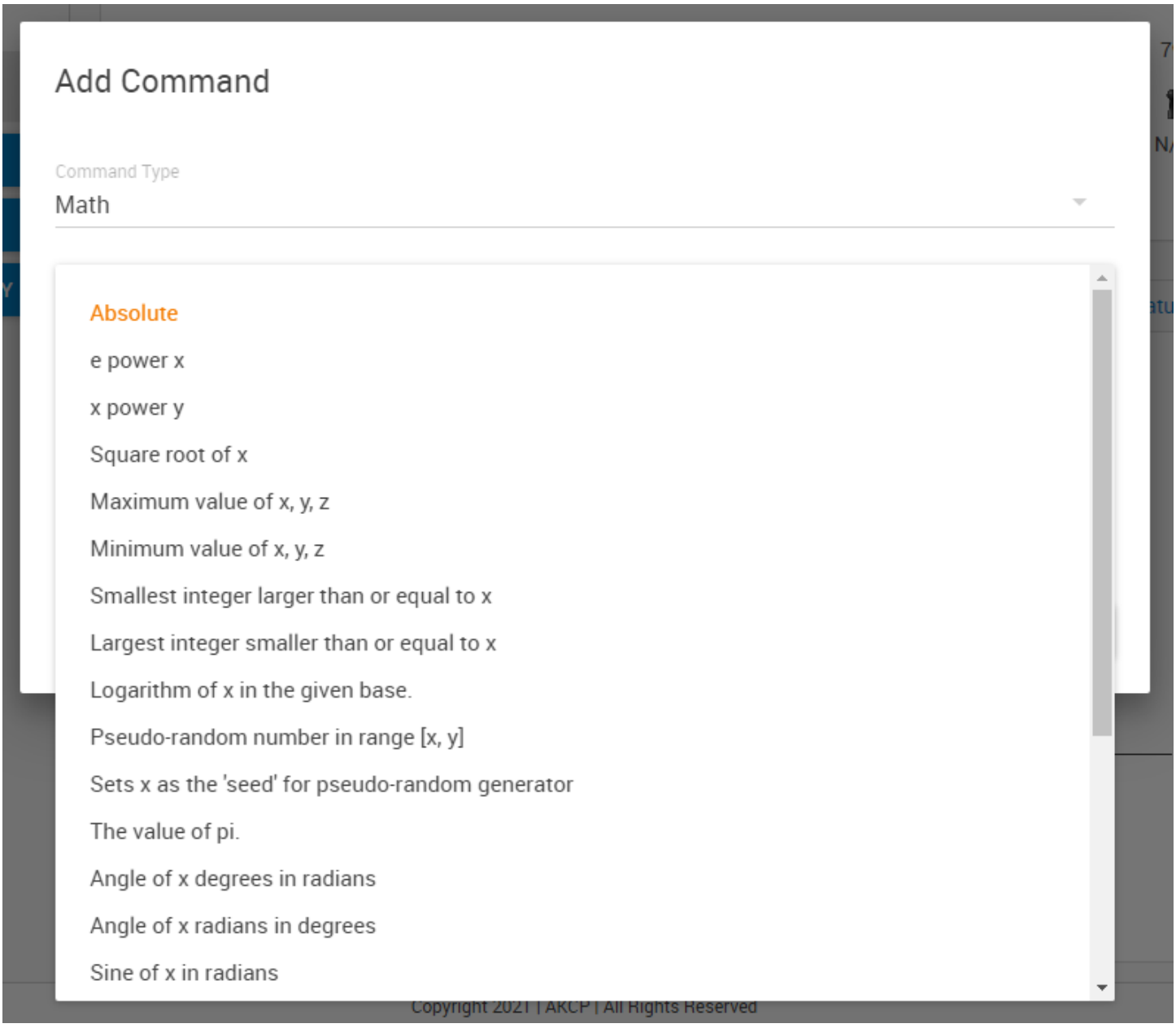


The **Command Function**'s list will depend on the selected Command Type. For example, for the Sensors you could select the following:



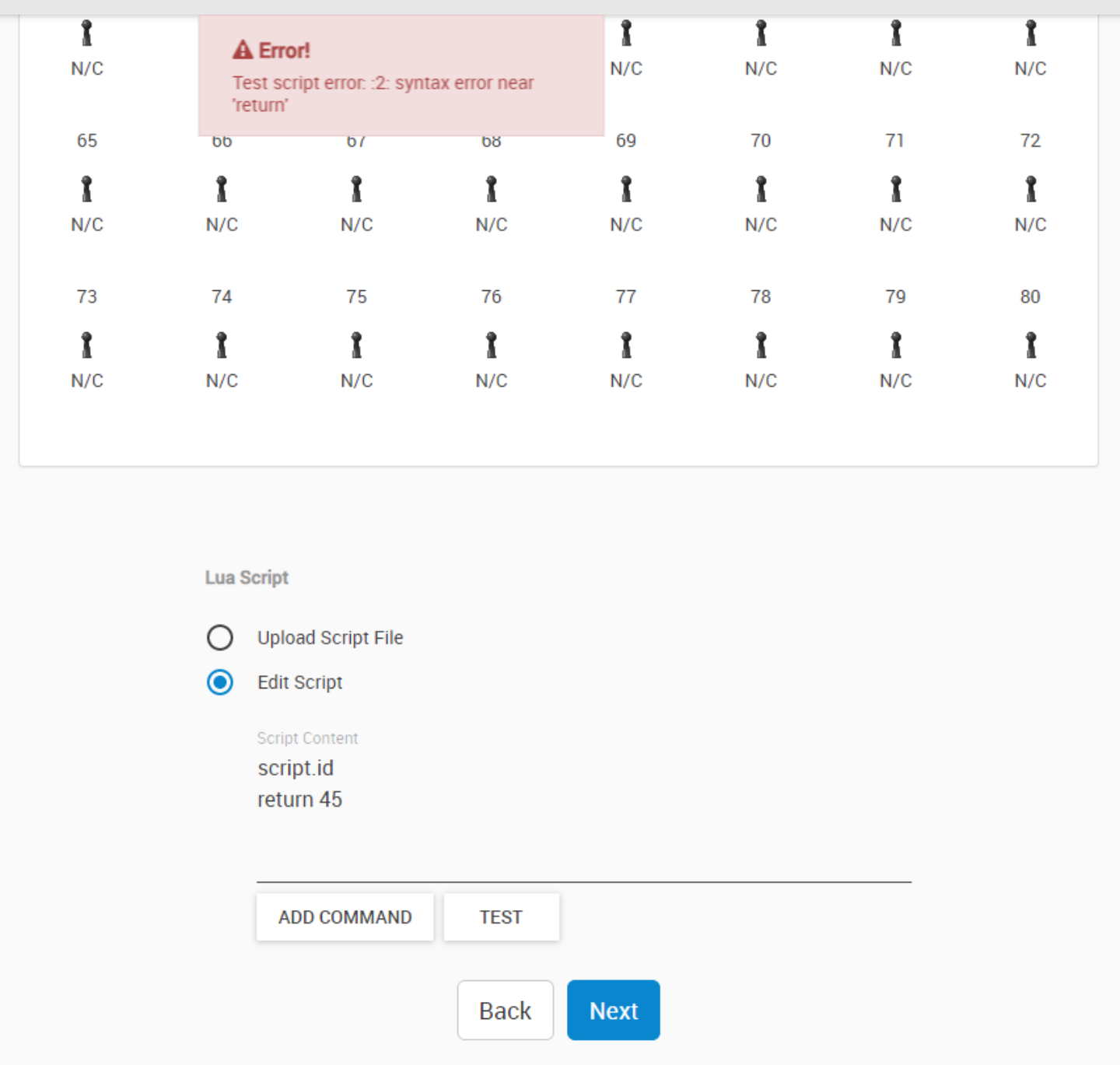
Please refer to the "*List of all available commands*" section at the end of this manual for the list of all available choices and combinations.

As another example, by choosing Math as Command Type, you will be given the Command Function selection as a list of possible math operations:

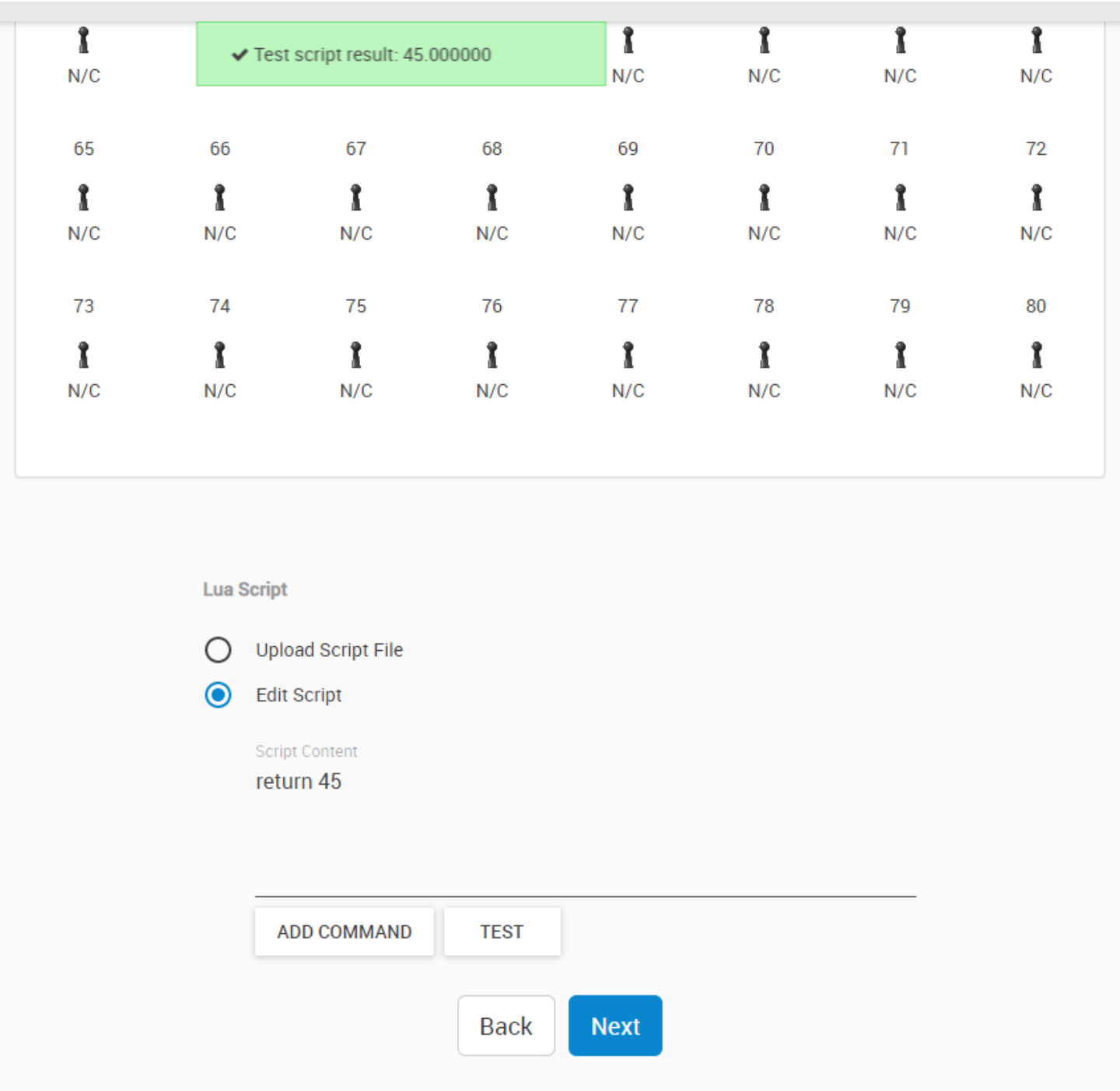


When you are done editing, click on the **Test** button to check your script.

If there is any syntax errors or other issues, the WebUI will display a warning popup and a hint on which line the error is:



If there are no errors with your script, the script's return value will be displayed in a green popup:

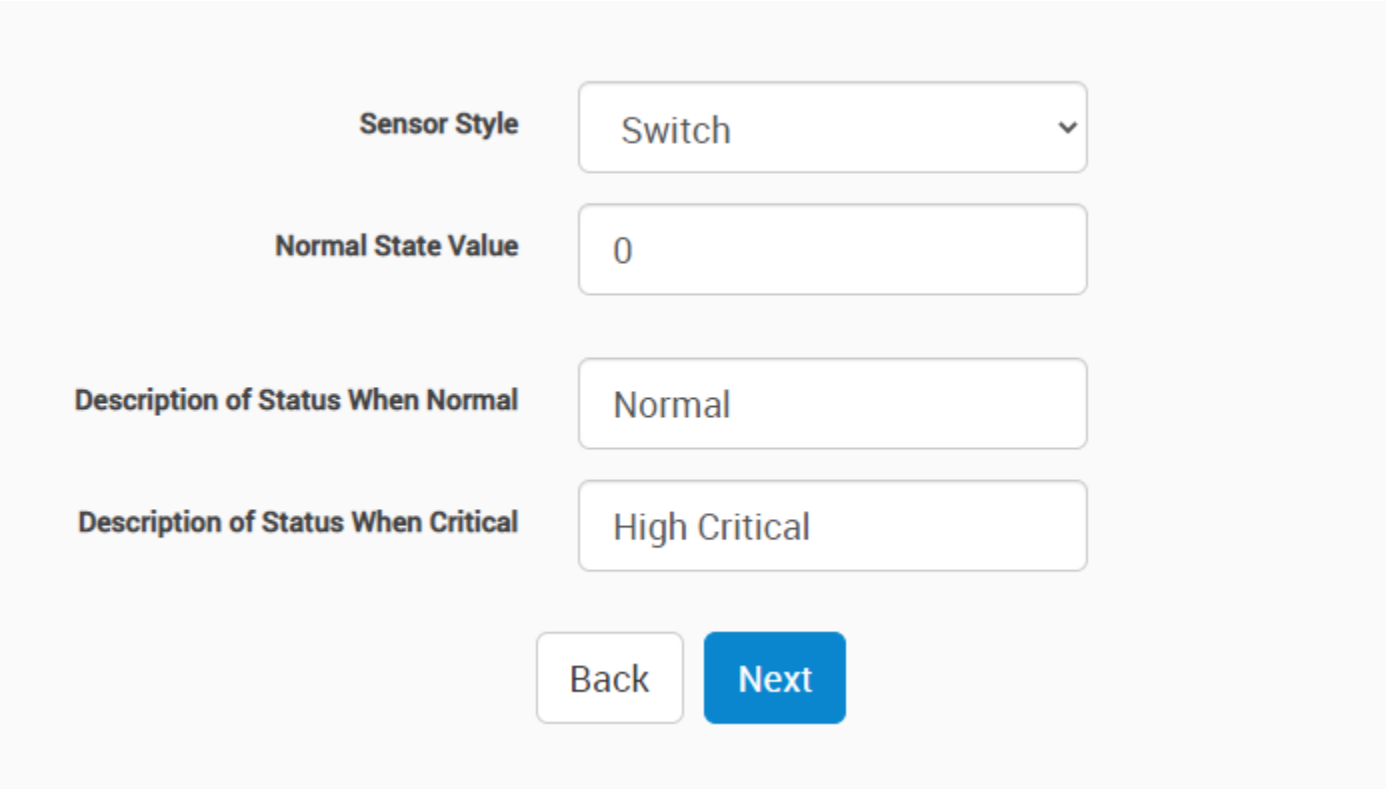


Make sure to always test your script before saving it!

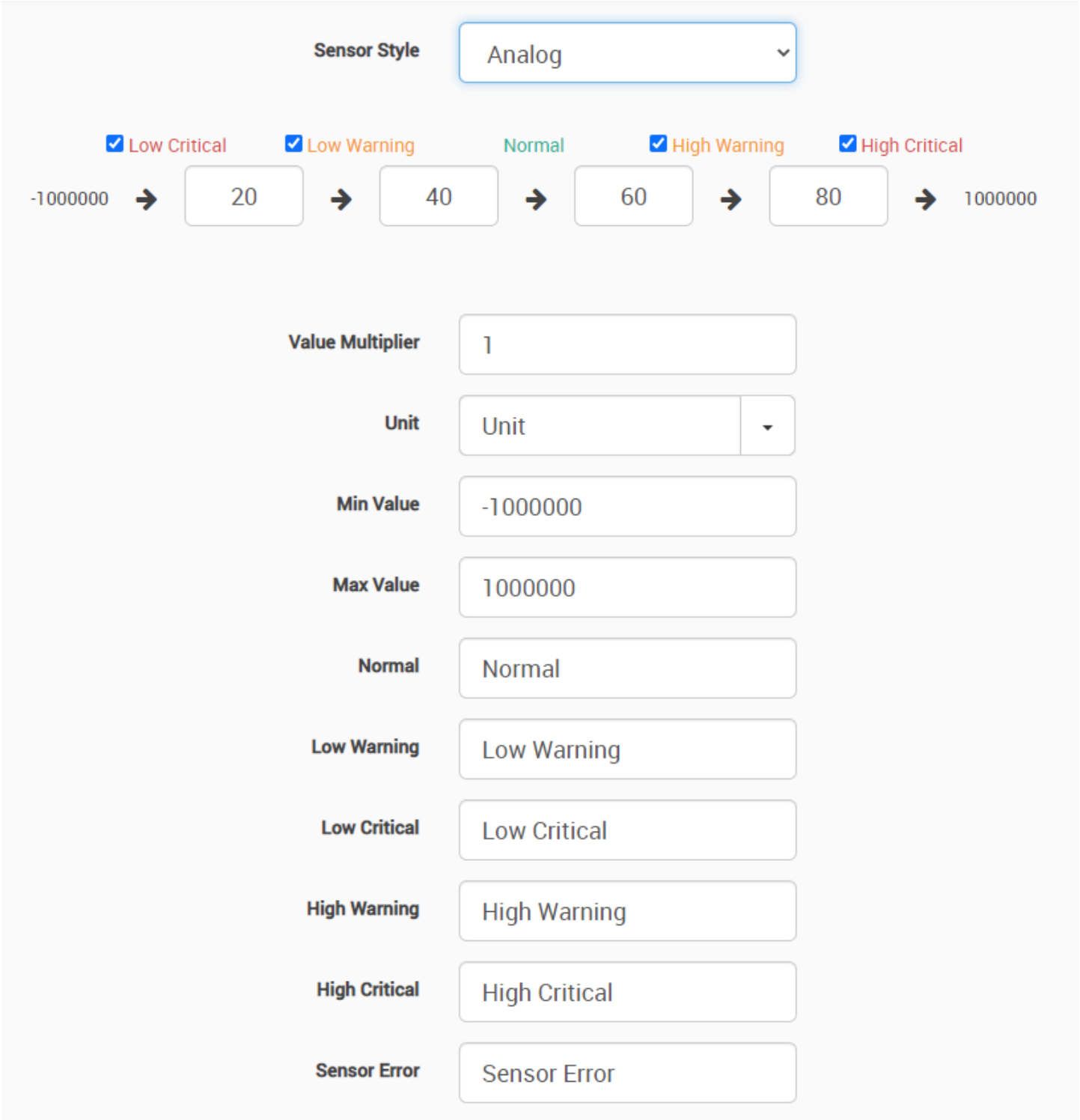
Next, choose the **Sensor Style** between Switch or Analog.
These options are similar to other AKCP sensors and therefore we will not detail all settings in this manual.

You would always need to choose the appropriate Sensor Style depending on your script's output and usage.

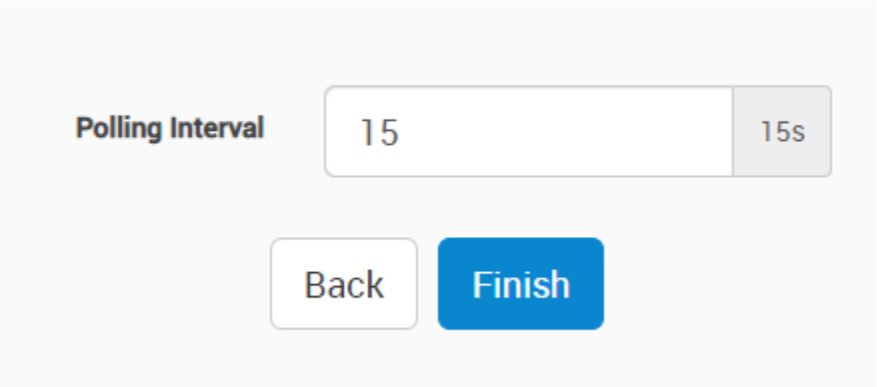
You could choose Switch style for example if your script will return a fixed number.
Set the Normal State Value to this expected number and your custom script's status will be Normal.
Any other return value that the script gets will result in the VS to be High Critical.



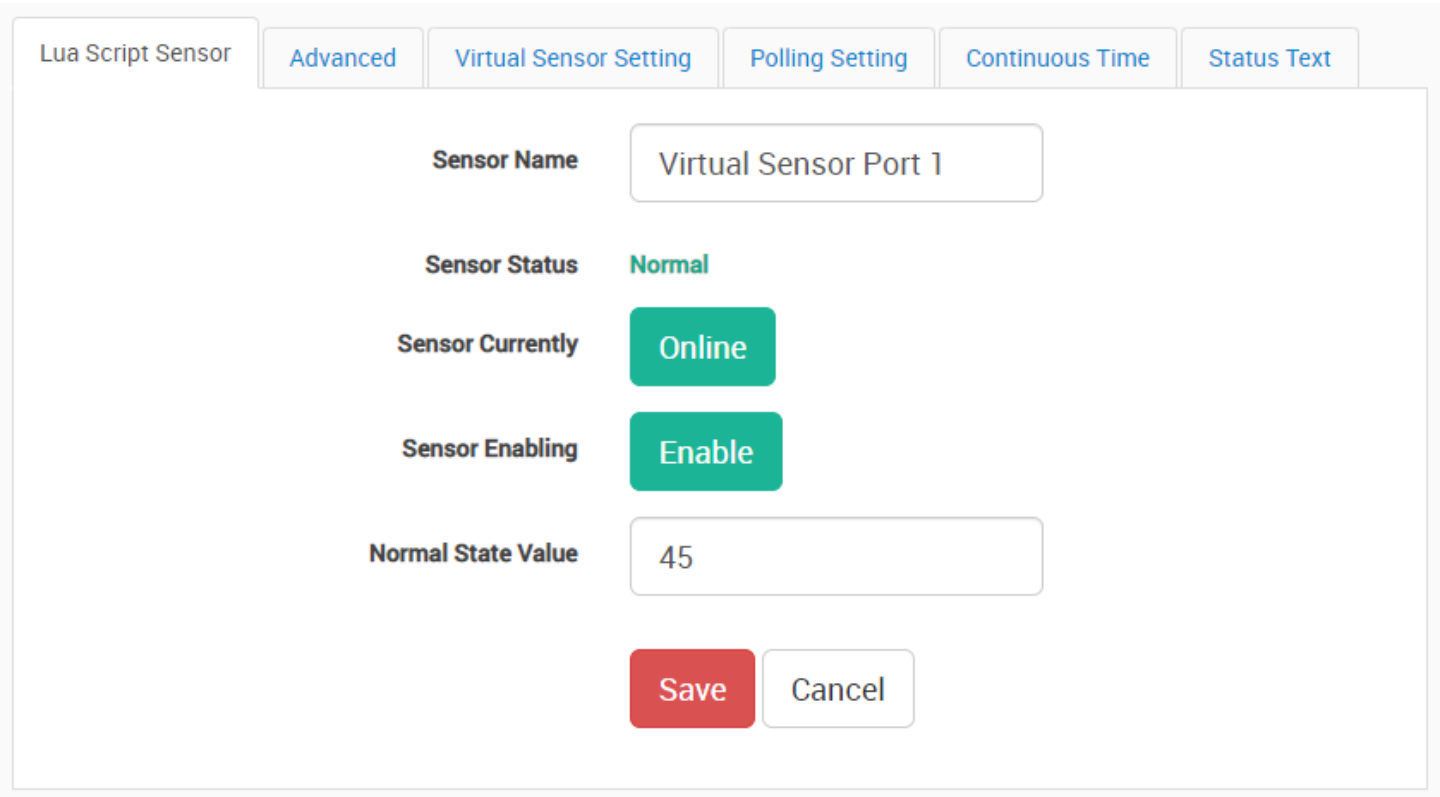
Choose the Analog style if you would want to have multiple statuses depending on the script's output. Modify the thresholds accordingly.



Set the **Polling Interval** as the final step. This defines how often the script will be executed. The default is 15 seconds.



The script will display the current status or reading, depending on your chosen Sensor Style:



Script examples

3 sensor values calculation

The following example script will do calculations on 3 sensor values:

```
-- Enter your preferred Virtual Sensors here

virtualSensorPartA = 1
virtualSensorPartB = 2
virtualSensorPartC = 3

------------------------------------------

-- Value preparation

partAValue = vsens.find(virtualSensorPartA).value
partBValue = vsens.find(virtualSensorPartB).value
partCValue = vsens.find(virtualSensorPartC).value

-- Calculation starts here

partialValue = partAValue + partBValue + partCValue

if partialValue == 0 then
    return -100
else
    return partialValue
end
```

You may modify the last part of the script in this way, if the check should result in sensor error status when the partialValue is 0:

```
if partialValue == 0 then
    error ('partialValue is ' .. partialValue)
else
    return partialValue
end
```

PUE value calculation

The following example script will do calculations for PUE with 3 sensor values (the partialValue is considered the IT power consumption):

```
-- Enter your preferred Virtual Sensors here

virtualSensorTotal = 1
virtualSensorPartA = 2
virtualSensorPartB = 3

-- Value preparation

totalValue = vsens.find(virtualSensorTotal).value
partAValue = vsens.find(virtualSensorPartA).value
partBValue = vsens.find(virtualSensorPartB).value

-- Calculation starts here

partialValue = partAValue + partBValue

if partialValue == 0 then
return -100
end

pue = totalValue / partialValue

return pue
```

The following example script is a modification to the earlier simpler PUE script, adding further calculations to the PUE:

```
-- Enter your preferred Virtual Sensors here

virtualSensorTotal = 1
virtualSensorPartA = 2
virtualSensorPartB = 3

-----------------------------------------

-- Value preparation

totalValue = vsens.find(virtualSensorTotal).value
partAValue = vsens.find(virtualSensorPartA).value
partBValue = vsens.find(virtualSensorPartB).value

-- Calculation starts here

partialValue = partAValue + partBValue

if partialValue == 0 then
    return -100
end

pueInt = math.floor(totalValue / partialValue)
rest = totalValue - (pueInt * partialValue)
pue = ((pueInt * 100) + (rest * 100) / partialValue)

return pue
```

Advanced continuous time calculation

The following example script will do calculations for the continuous time:

```
-- Calculate continuous time, in seconds, when the value of Virtual sensor 1 is
greater than 100
value = vsens.find(1).value
threshold = 100

if value < threshold then
    storage.reset('t100')
    return 0
end

start_time = storage.get('t100')
if not start_time then
    storage.set('t100', os.time())
    start_time = os.time()
end
return os.time() - start_time
```

For more scripting examples, please refer to the "*List of all available commands*" section at the end of this manual.

List of all Lua functions provided by SP+ device

```
sensor.find(id) -> { id, status_update_time, value, status, acknowledge }
vsens.find(index) -> index=1-80
os.time()
os.clock()
script.id
script.run(id)
storage.id
storage.clear()
storage.reset(key)
storage.set(key, value)
storage.setTimer(key, expireMs)
storage.get(key) -> value
storage.timerExpired(key) -> bool
```

For detailed descriptions of the functions with scripting examples, please refer to the "*List of all available commands*" section at the end of this manual.

List of all available commands for Add Command wizard

Command Type	Command Function	Command syntax	Description	Example Usage
Sensor	*Reading*	sensor.find(<SensorID>).value vsens.find(<VirtualSensorIndex>).value	Get sensor reading (value)	-- Return value of sensor port 1 subport 1 as Lua Virtual Sensor value return sensor.find(0).value
Sensor	*Status*	sensor.find(<SensorID>).status vsens.find(<VirtualSensorIndex>).status	Get sensor status	-- Return status of sensor port 1 subport 2 as Lua Virtual Sensor value return sensor.find(1).status
Sensor	*Acknowledge*	sensor.find(<SensorID>).acknowledge vsens.find(<VirtualSensorIndex>).acknowledge	Get sensor status "Acknowledged" flag. When sensor status turned Warning / Critical, user can manually acknowledge the sensor status in SPX+ summary page by pressing "..." >>> Acknowledge on the desired sensor. Value: 0 if the sensor status is not acknowledged (default), 1 if it is.	-- Return sensor status "Acknowledged" flag of Virtual Sensor 1 as Lua Virtual Sensor value return vsens.find(1).acknowledge
Sensor	*Last status change timestamp*	sensor.find(<SensorID>).status_update_time vsens.find(<VirtualSensorIndex>).status_update_time	Get sensor last status update time	-- Store update time of sensor port 2 subport 1 into a variable -- This can be used for further calculation lastUpdateTime = sensor.find(257).status_update_time
Sensor	*ID*	sensor.find(<SensorID>).id vsens.find(<VirtualSensorIndex>).id	Get sensor ID	-- Get sensor ID of Virtual Sensor 10 -- This can be used for further calculation virtualSensorId = vsens.find(10).id

Command Type	Command Function	Command syntax	Description	Example Usage
Operating system	*Operating system timestamp*	os.time()	Get current UNIX timestamp	-- Check if it is working hours by UTC local utc_hour = os.time() / (60*60) % 24 return 10 <= utc_hour and utc_hour <= 18
Operating system	*Operating system clock*	os.clock()	Get time from device start, in milliseconds	-- The sensor will show the device uptime return os.clock()

Command Type	Command Function	Command syntax	Description	Example Usage
Script	*Script ID*	script.id	Get current Lua script ID. Equals to Lua virtual sensor index.	-- Return previous value return vsens.find(script.id).value
Script	*Script Execute*	script.run(<LuaScriptID>)	Run Lua script by ID	-- Reuse another Lua script return script.run(40)

Command Type	Command Function	Command syntax	Description	Example Usage
Storage	*ID*	storage.id	Get current storage ID. Equals to Lua virtual sensor index.	-- Return previous value return vsens.find(storage.id).value
Storage	*Clear*	storage.clear()	Remove all the keys from current storage	storage.clear()
Storage	*Reset*	storage.reset(<Key>)	Remove selected key from current storage	storage.reset('myKey')
Storage	*Set*	storage.set(<Key>,<Value>)	Save value to key	-- Set 'myKey' to 50 storage.set('myKey', 50)
Storage	*Get*	storage.get(<Key>)	Get saved value	-- Get 'myKey' value, or 100 if 'myKey' was not set yet -- Value is available even if set on a previous Lua run return storage.get('myKey') or 100
Storage	*Set timer*	storage.setTimer(<TimerKey>,<TimeMs>)	Set timer to key	-- Timer will be expired in 1 second storage.setTimer('myTimer', 1000)
Storage	*Check timer expired*	storage.timerExpired(<TimerKey>)	Check if timer is expired	return storage.timerExpired('myTimer')

Command Type	Command Function	Command syntax	Description	Example Usage
Math	*Absolute*	math.abs(<Value>)	Return the absolute, or non-negative value, of a given value.	> = math.abs(-100) 100 > = math.abs(25.67) 25.67 > = math.abs(0) 0
Math	*e power x*	math.exp(<x>)	math.exp(x) returns e (the base of natural logarithms) raised to the power x.	> = math.exp(0) 1 > = math.exp(1) 2.718281828459 > = math.exp(27) 532048240601.8
Math	*x power y*	x ^ y	Return x power by y	> = 2 ^ 10 1024 > = 10 ^ -1 0.100000 > = -5 ^ -3.123 -0.006563
Math	*Square root of x*	math.sqrt(<x>)	Return the square root of a given number. Only non-negative arguments are allowed.	> = math.sqrt(100) 10 > = math.sqrt(1234) 35.128336140501 > = math.sqrt(-7) -1.#IND
Math	*Maximum value of x,y,z*	math.max(<x>, <y>, <z>)	Return the maximum value from a variable length list of arguments.	> = math.max(1.2, -7, 3) 3 > = math.max(1.2, 7, 3) 7 math.modf
Math	*Minimum value of x, y, z*	math.min(<x>, <y>, <z>)	Return the minimum value from a variable length list of arguments.	> = math.min(1,2) 1 > = math.min(1.2, 7, 3) 1.2 > = math.min(1.2, -7, 3) -7

Math	*Smallest integer larger than or equal to x*	math.ceil(<Value>)	Return the integer no greater than or no less than the given value (even for negatives)	> = math.floor(0.5) 0 > = math.ceil(0.5) 1 > = math.floor(-0.5) -1 > = math.ceil(-0.5) -0
Math	*Largest integer smaller than or equal to x*	math.floor(<Value>)		
Math	*Logarithm of x in the given base*	math.log(<x> [, base])	Returns the logarithm of x in the given base. The default for base is e (so that the function returns the natural logarithm of x).	> = math.log(532048240601) 26.999999999998 > = math.log(3) 1.0986122886681 > = math.log(100, 10) 2.000000
Math	*Returns the integral fractional parts of x*	math.modf(<x>)	Returns the integral part of x and the fractional part of x. Its second result is always a float.	modResult = {math.modf(-5.3)} -- The function returns 2 values (integral and fractional parts) and it can be accessed as follows return modResult[1] -- << -5.000000 return modResult[2] -- << -0.300000
Math	*Pseudo-random number in range [x, y]*	math.random([x [, y]])	When called without arguments, returns a pseudo-random float with uniform distribution in the range [0,1). When called with two integers m and n, math.random returns a pseudo-random integer with uniform distribution in the range [m, n]. (The value n-m cannot be negative and must fit in a Lua integer.) The call math.random(n) is equivalent to math.random(1,n). This function is an interface to the underling pseudo-random generator function provided by C.	> = math.random() 0.0012512588885159 > = math.random() 0.56358531449324 > = math.random(100) 20 > = math.random(100) 81 > = math.random(70,80) 76 > = math.random(70,80) 75
Math	*Sets x as the 'seed' for pseudo-random generator*	math.randomseed(<x>)	Sets x as the "seed" for the pseudo-random generator: equal seeds produce equal sequences of numbers.	math.randomseed(1234) return math.random() -- << always 0.915030
Math	*The value of pi*	math.pi	The value of π.	> = math.pi 3.1415926535898

Math	*Angle of x degrees in radians*	math.rad(<x>)	Convert from degrees to radian.	> = math.rad(180) 3.1415926535898 > = math.rad(1) 0.017453292519943
Math	*Angle of x radians in degrees*	math.deg(<x>)	Convert from radian to degrees	> = math.deg(math.pi) 180 > = math.deg(math.pi / 2) 90
Math	*Sine of x in radians*	math.sin(<x>)	Return the cosine, sine and tangent value for a given value in radians.	> = math.cos(math.pi / 4) 0.70710678118655 > = math.sin(0.123) 0.12269009002432 > = math.tan(5/4) 3.0095696738628 > = math.tan(.77) 0.96966832796149
Math	*Cosine of x in radians*	math.cos(<x>)		
Math	*Tangent of x in radians*	math.tan(<x>)		
Math	*Arc sine of x in radians*	math.asin(<x>)	Return the inverse cosine and sine in radians of the given value.	> = math.acos(1) 0 > = math.acos(0) 1.5707963267949 > = math.asin(0) 0 > = math.asin(1) 1.5707963267949
Math	*Arc cosine of x in radians*	math.acos(<x>)		

Math	*Arc tangent of y/x in radians*	math.atan(y [, x])	Return the inverse tangent in radians. We can do this by supplying y/x ourselves or we can pass y and x to math.atan to do this for us.	> c, s = math.cos(0.8), math.sin(0.8) > = math.atan(s/c) 0.8 > = math.atan(s,c) 0.8 > = math.atan(10) 1.471128
Math	*Value larger than or equal to any other numerical value*	math.huge	The float value HUGE_VAL, a value larger than any other numeric value.	> = math.huge inf > = math.huge / 2 inf > = -math.huge -inf > = math.huge/math.huge -- indeterminate nan > = math.huge * 0 -- indeterminate nan > = 1/0 inf > = (math.huge == math.huge) true > = (1/0 == math.huge) true
Math	*Remainder of the division of x by y that rounds the quotient towards zero*	math.fmod(<x>, <y>)	Returns the remainder of the division of x by y that rounds the quotient towards zero. (integer/float)	> = math.fmod(10, 2) 0 > = math.fmod(-5, 3) -2.000000 > = math.fmod(6.6, 2.5) 1.600000

Please contact support@akcp.com if you have any further technical questions or problems.

Thanks for Choosing AKCP!