xxter Lua scripts manual

Version 1.6 – April 2023

xxter scripts introduction

With xxter scripts you can create your own small programs within xxter. These scripts are very flexible and can be used to add a wide variety of features to a home or building automation installation. You can create logic functions, delay actions, extend your scenarios with RGB sequences and much more. Scripts can be activated using a scenario, a schedule, an action (trigger) or another script.

xxter supports both a native scripting language as well as Lua scripting support. This manual provides explanation on Lua scripts. For xxter native scripts, please see the documentation page of our website (https://www.xxter.com/documentation/). We also have several examples available on this page and also on our forum (https://forum.xxter.com).

To learn more about Lua in general, please refer to the Lua manual:
https://www.Lua.org/manual/5.3/

The basics

xxter Lua scripts are written programs that consist of one or more lines. You can add comments to a script by writing "--". This can be done on a separate line or at the end of a command.

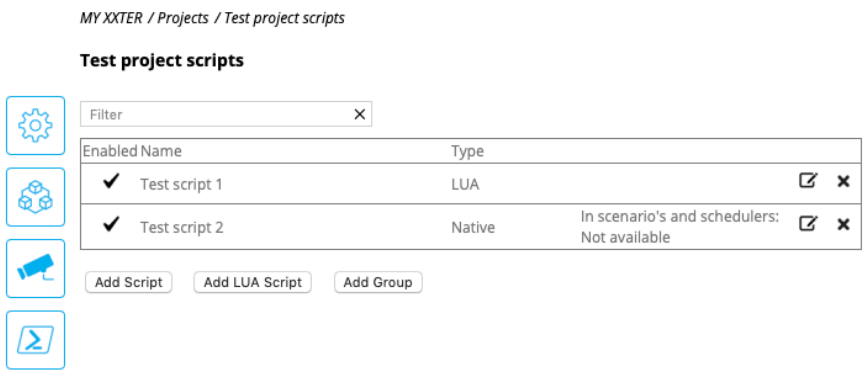
An example of an xxter Lua script:

```
1 -- this is an example test script.
2 xxter.userlog("starting test script")
3 -- Setting dimmer to 50%
4 xxter.setcomponent(16, 50)
5 -- Waiting for 5 seconds
6 xxter.sleep(5000)
7 -- Setting switch on
8 xxter.setcomponent(56, 1)
9
```

Scripts can be started and stopped by many means, for example by a scenario or a schedule. Scripts can be "unending" scripts that perform certain actions in repeating intervals or can be defined as a sequence of commands that are executed only once, whenever the script is started. When using "unending" scripts that are continuously looped, please remember to always include a "sleep" period to prevent the script creating a very heavy load on the xxter controller.

Script management

xxter scripts can be created and updated using the xxter editor. This editor can be found in the project configuration of “My xxter”. Select the project for which you want to manage the scripts and select “Scripts” in the menu bar. All existing scripts will be shown here and can be edited and deleted. When creating a new script, you can choose whether you want to create a native xxter script, or a Lua script.



Important: When you add or edit a script, this has to be loaded onto the xxter controller before it can be used.

Whenever a change has been made, you can see the text “Project has changed” at the top right-hand corner. When you click on this and subsequently on “Push configuration to xxter device”, the controller will download the new configuration. This feature works best when xxter is configured for use outside (see the installation manual). Alternatively, you can reload the profile using the app (see the user manual). Of course, you can also reload the project from the xxter controller itself, by logging onto the xxter controller and clicking the button “Load configuration” at the top left-hand corner in the menu.

When you are logged in on the xxter controller, you can see the loaded scripts by clicking the “Scripts” option in the menu. Here a script can also be manually started, restarted or stopped for testing purposes.

Scripts

To change actions and scripts, go to 'My xxter'.

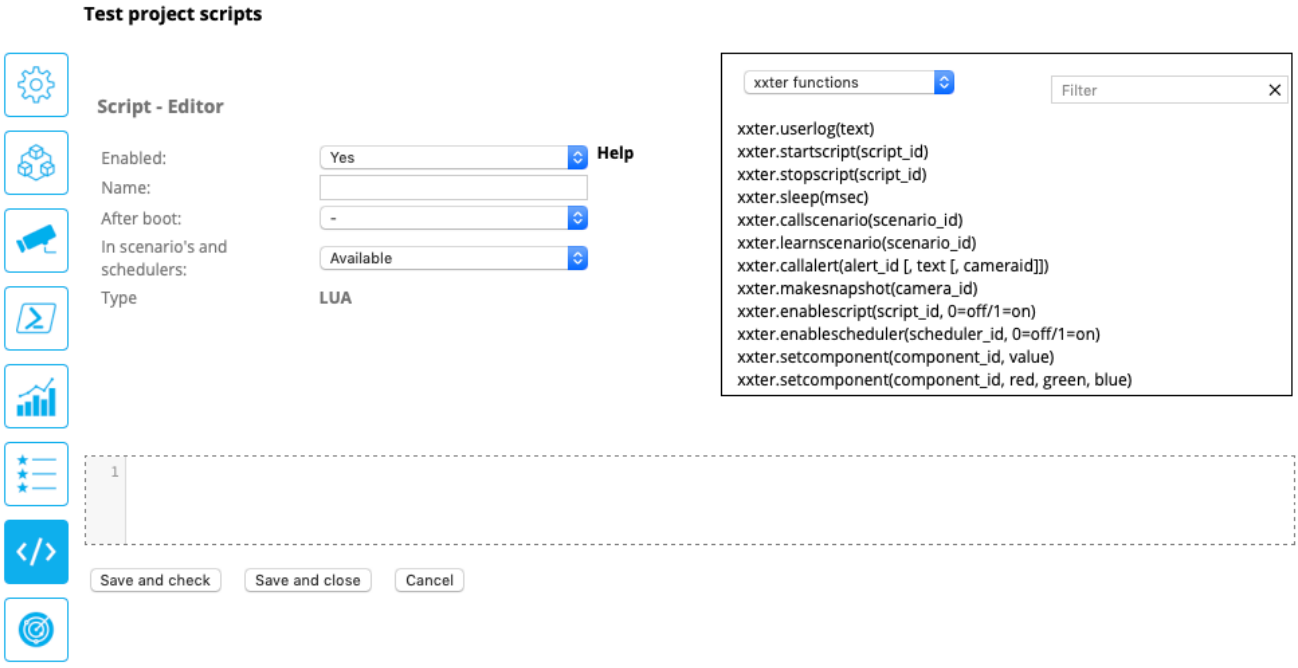
Active	Name		Status	
Yes	Testscript 1		Stopped	Start Restart Stop
No	Testscript 2	In scenario's and schedulers: Not available	Not active	Start Restart Stop

For troubleshooting your scripts, the device logs are very helpful. On the xxter controller, you can enable “Scripts” under User log on the *Settings – basic* page. This will log every executed line of every script in the device log. You can access the log via “Open user log window” on the *Status* page.

Script editor

When you add or edit a script, you can select whether it should be enabled and you can select whether the script should be available for the end user, to use in scenarios or the scheduler. You can also select if the script should be started automatically, whenever the xxter controller is (re) started.

IMPORTANT: Disabled scripts cannot be executed and will not be triggered when included in a scenario, action or scheduler. This can be useful for testing and troubleshooting proposes. However, keep in mind that a script can be enabled or disabled by another script!



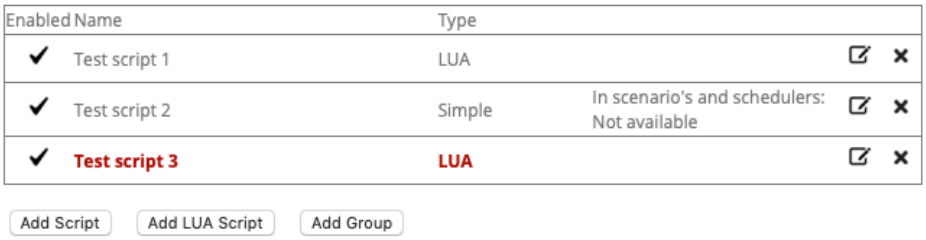
The line numbers that are displayed next to the script in the editor are informational only and are not used in the scripts themselves. Commands can be added by typing them in directly or by using the command selection tool, located at the right-hand side top corner. Make sure you are at the right position in the script editor when adding a command with the commands tool. When adding a command with the commands tool, the variables need to be set to the appropriate values. The functions will provide information what kind of values need to be added. The IDs of components can be selected from the selection tool, by selecting to type of value you need (e.g. “scripts” or “bytes”). Clicking on a component, will add this to the script at the cursor location.

After you have added one or more commands in the editor, you can verify if they are valid by clicking the "Save and check" button. Your current script will be checked and redisplayed in the editor. If there is a syntax error in the script, a message will be shown above the edit screen. However parsing errors (e.g. validating whether functions exist or have the right amount and type of variables) cannot be detected in the editor.

4: unexpected symbol near '5'

```
1 -- this is another example
2 xxter.startscript(5367)
3 -- here is a syntax error
4 5 + fault
5 -- but please note that parsing errors are not detected here
6 this.functiondoesnotexist()
7
```

After correcting or adding lines, you can easily verify the script again by using the "Save and check" button. When everything is correct you can use the "Save and close" button to close the editor. Scripts that are saved but contain an error are shown in red.



Note: scripts that contain errors can be saved but will not start.

Scripts will automatically stop running after the last command of the script has been executed.

Scripts are written using functions and values. Additionally, scripts can be extended using variables, and control structures like IF statements and WHILE loops. It is also possible to define your own functions, with input parameters and return values.

To learn more about Lua, please refer to the Lua manual: https://www.Lua.org/manual/5.3/

xxter functions

The following specific xxter Lua functions are available.

---

xxter.userlog(text)

Description:
Writes to the user log, useful for debugging your Lua scripts. Writing to the user log this way will always take place, even when Script logging has been disabled on the *Settings – basic* page of the device, see page 3 of this manual.

Parameters:
text : a string with the text to write to the user log.

Returns: -

Example:
```
xxter.userlog("Running script part X")
```

---

xxter.startscript(script_id)

Description:
Starts an xxter script, if it is not already running.

Parameters:
script_id : The identifier of the script (number)

Returns: -

Example:
```
xxter.startscript(35)
```

---

xxter.stopscript(script_id)

Description:
Stops an xxter script.

Parameters:
script_id : The identifier of the script (number)

Returns: -

Example:
```
xxter.stopscript(35)
```

---

xxter.enablescript(script_id, status)

Description:
Enables or disables an xxter script, so it can (not) be started.

Parameters:
script_id : The identifier of the script (number)
status : Desired script status (enum: 0 = off, 1 = on)

Returns: -

Example:
```
xxter.enablescript(35,1)
```

---

xxter.include(script_id)

Description:
Includes all the code of the designated script into the current script. This, for instance, allows you to define a set of functions in one script and use that same code in multiple scripts.

Parameters:
script_id : The identifier of the script (number)

Returns: -

Example:
```
xxter.include(17)
```

---

xxter.sleep(time)

Description:
Waits for the given amount of time (in milliseconds), before continuing.

Parameters:
time : The time period in milliseconds to wait (number)

Returns: -

Example:
```
xxter.sleep(5000)
```

---

xxter.callscenario(scenario_id)

Description:
Calls the specified xxter scenario to perform its configured actions.

Parameters:
scenario_id : The identifier of the scenario (number)

Returns: -

Example:
```
xxter.callscenario(17)
```

xxter.learnscenario(scenario_id)

Description:
Updates the configured actions of the specified xxter scenario to their current values

Parameters:
scenario_id : The identifier of the scenario (number)

Returns: -

Example:
```
xxter.learnscenario(17)
```

---

xxter.callalert(alert_id[, text [,camera_id]])

Description:
Calls the specified xxter alert, optionally with specific text and camera snapshot. The (optional) text that is provided, is placed in the "[x]" position of the alert service, if available.

Parameters:
alert_id : The identifier of the alert (number)
text : *Optional,* the text to be sent with the alert (string)
camera_id : *Optional,* the identifier of the camera to take a snapshot (number)

Returns: -

Examples:
```
xxter.callalert(65)
xxter.callalert(65, "Scripted alert")
xxter.callalert(65, "Scripted alert", 103)
```

---

xxter.makesnapshot(camera_id)

Description:
Makes a snapshot of the provided camera.

Parameters:
camera_id : The identifier of the camera to take a snapshot (number)

Returns: -

Examples:
```
xxter.makesnapshot(103)
```

---

xxter.enablescheduler(scedule_id, status)

Description:
Enables or disables an xxter scheduler, so it can (not) be started.

Parameters:
scedule_id : The identifier of the schedule (number)
status : Desired scheduler status (enum: 0 = off, 1 = on)

Returns: -

Example:
```
xxter.enablesceduler(18,1)
```

---

xxter.setcomponent(component_id, value [, value_2, value_3])

Description:
Sets a component to the provided value.

Parameters:
component_id : The identifier of the component (number)
value : Desired value of the component (type depends on component).
In case component is RGB, this value refers to R (number)
value_2 : *Only for RGB,* refers to G (number)
value_3 : *Only for RGB,* refers to B (number)

Returns: -

Examples:
```
xxter.setcomponent(38,75)
xxter.setcomponent(45,75,230,176)
```

---

xxter.getcomponent(component_id)

Description:
Gets the current value of a component.

Parameters:
component_id : The identifier of the component (number)

Returns:
value : Current value of the component (type depends on component).
RGB components, return three values

Example:
```
var = xxter.getcomponent(38)
varR, varG, varB = xxter.getcomponent(45)
```

---

xxter.readcomponent(component_id)

Description:
Performs a read command on the KNX bus of the component. If the KNX component responds to the request, xxter will update its current status (this may take some time). Subsequently, the current value can be used in the script with *xxter.getcomponent*).

Parameters:
component_id : The identifier of the component (number)

Returns: -

Example:
```
xxter.readcomponent(38)
```

---

xxter.httpcommand(http_id [, result])

Description:
Executes the provided HTTP command.

Parameters:
http_id : The identifier of the http command (number)
result : Optional parameter to also request the command result (boolean)

Returns:
result : If requested and available, the result that is returned by the HTTP command (string)

Example:
```
xxter.httpcommand(12)
var = xxter.httpcommand(13,true)
```

---

xxter.tcpcommand(tcp_id)

Description:
Executes the provided TCP command.

Parameters:
tcp_id : The identifier of the tcp command (number)

Returns: -

Example:
```
xxter.tcpcommand(17)
```

---

xxter.ircommand(ir_id)

Description:
Executes the provided infra red command.

Parameters:
ir_id : The identifier of the IR command (number)

Returns: -

Example:
```
xxter.ircommand(7)
```

---

xxter.openKNXtunnel(status)

Description:
Opens or closes the KNX tunnel.

Parameters:
status : Command to open or close the tunnel (enum; 0=close, 1=open)

Returns: -

Example:
```
xxter.openKNXtunnel(1)
```

---

xxter.setsimulation(status)

Description:
Sets the presence simulation to the desired state.

Parameters:
status : Desired state of the presence simulation (enum; 0=stop, 1=record, 2=play)

Returns: -

Example:
```
xxter.setsimulation(2)
```

---

xxter.getsimulation()

Description:
Gets the current state of the presence simulation.

Parameters: -

Returns:
Current state of the presence simulation (enum; 0=stop, 1=record, 2=play)

Example:
```
var = xxter.getsimulation()
```

---

xxter.connectKNX(status)

Description:
Connect or disconnects the KNX connection of the device.

Parameters:
status : Command for the KNX connection (enum; 0=disconnect, 1=connect)

Returns: -

Example:
```
xxter.connectKNX(1)
```

---

xxter.setpersistent(varname, value)

Description:
Create or update a variable that is persistent over all scripts, to pass information between scripts.

Parameters:
varname : Name of persistent variable (string, alphanumerical: 0-9a-zA-Z)
value : Value the persistent variable should have (type depending on the variable)

Returns: -

Examples:
```
xxter.setpersistent("maxValue",21.23)
xxter.setpersistent("message","Valve error")
```

---

xxter.getpersistent(varname)

Description:
Retrieve a variable that is persistent over all scripts, to pass information between scripts.

Parameters:
varname : Name of persistent variable (string, alphanumerical: 0-9a-zA-Z)

Returns:
Value of the persistent variable (type depending on the variable)

Examples:
```
var = xxter.getpersistent("maxValue")
```

---

xxter.getSunAzimuth()

Description:
Get the current sun angle, in relation to north (direction), based on the location configured on the xxter controller.

Parameters: -

Returns:
Value of the direction of the sun.

Examples:
```
var = xxter.getSunAzimuth()
```

---

xxter.getSunAltitude()

Description:
Get the current sun angle, in relation to the horizon (altitude), based on the location configured on the xxter controller.

Parameters: -

Returns:
Value of the altitude of the sun.

Examples:
```
var = xxter.getSunAltitude()
```

---

xxter.setsonos(device_id, command [, option_setting])

Description:
Executes a Sonos command

Parameters:
device_id : Can be set either to “global” for global commands, or to provide a specific Sonos device identifier (format: RINCON_949F3EC192E401400).
command: when device_id is set to “global” the following commands can be used:

- “creategroup” : requires option_setting for groupid (number)
- “muteall”
- “unmuteall”
- “pauseall”
- “playall”

when the device_id of a Sonos device is set, the following commands can be used:

- “groupvolume” : requires option_setting for volume (number)
- “groupmute”
- “volume” : requires option_setting for volume (number)
- “mute”
- “unmute”
- “play”
- “pause”
- “seek” : requires option_setting for position (number)

"next"
"previous"
"selectplaylist" : requires option_setting for name of playlist (string)
"selectfavorite" : requires option_setting for name of favorite (string)
"selectnextfavorite"
"selectlinein"
"selectHDMI"
"playaudioclip" : requires two option_settings, one for type of audio clip (enum) and one for the clip number (number). Audio clip types are:
0: Sonos standard
1: xxter standard
2: custom

optional_setting : depending on the command used (see command parameters above)

Returns: -

Examples:
```
xxter.setsonos("global", "creategroup", 3)
xxter.setsonos("global", "muteall")
xxter.setsonos("RINCON_949F3EC192E401400", "selectfavorite", "Radio 4")
xxter.setsonos("RINCON_949F3EC192E401400", "playaudioclip", 2, 4)
```

Example 1 and 2 are global commands, to create a speaker group or to mute all speakers.
Example 3 and 4 are for one speaker, to play "Radio 4" or play a custom audio clip.

---

xxter.getsonos(device_id, parameter)

Description:
Request the current status of a Sonos device

Parameters:
device_id : Specific Sonos device identifier (format: RINCON_949F3EC192E401400)
parameter : The parameter that is to be returned (select one of the following:
"groupvolume", "volume", "status", "mute", "groupmute", "meta")

Returns:
groupvolume : current volume of the Sonos group of the device (number)
volume : current volume of the Sonos device (number)
status : current status of the Sonos device (enum: 0:IDLE, 1:BUFFER, 2:PLAY, 3:PAUSED)
(group)mute : current (group) mute status (1 if muted, 0 if not)
a, b, c, d, e, f, g : meta data of the current playing title (multiple strings)
A = track_name
B = track_artist
C = album_name
D = album_artist
E = show_name
F = container_name
G = stream_info

Examples:
```
groupvol = xxter.getsonos("RINCON_949F3EC192E401400", "groupvolume")
a, b, c, d, e, f, g = xxter.getsonos("RINCON_949F3EC192E401400", "meta")
```

Example 1 requests the group volume, example 2 the meta data of the current track.

xxter.setupnp(device_id, command[, option_setting])

Description:
Executes a uPnP command

Parameters:
device_id : Specific uPnP device identifier (format: RINCON_949F3EC192E401400)
command: the following commands can be used:

"volume"	: requires option_setting for volume (number)
"mute"	
"unmute"	
"play"	
"pause"	
"seek"	: requires option_setting for position (number)
"next"	
"previous"	

optional_setting : depending on the command used (see command parameters above)

Returns: -

Examples:
```
xxter.setupnp("RINCON_949F3EC192E401400", "volume", 25)
xxter.setupnp("RINCON_949F3EC192E401400", "mute")
```

Example 1 sets the volume of the speaker, example 2 puts the speaker on mute.

---

xxter.getupnp(device_id)

Description:
Request the current status of a uPnP device

Parameters:
device_id : Specific uPnP device identifier (format: RINCON_949F3EC192E401400)
parameter : The parameter that is to be returned (select one of: "volume", "status", "meta")

Returns:
volume : current volume of the uPnP device (number)
status : current status of the uPnP device (string)
a, b, c, d, e, f, g : meta data of the current playing title (multiple strings)

A = track_name
B = track_artist
C = album_name
D = album_artist
E = show_name
F = container_name
G = stream_info

Examples:
```
volume = xxter.getupnp("RINCON_949F3EC192E401400", "volume")
a, b, c, d, e, f, g = xxter.getupnp("RINCON_949F3EC192E401400", "meta")
```

Example 1 requests the volume of the speaker, example 2 the meta data of the current track.

---

xxter.timezonediff()

Description:
Provides the difference in seconds between local time zone and UTC (GMT without daylight saving).

Parameters: -

Returns:
Time zone difference with UTC in seconds

Example:
```
var = xxter.timezonediff()
```

---

xxter.localtime()

Description:
Provides the local time in seconds. This works similar to os.clock() but then for the local time zone instead of UTC (GMT without daylight saving). Function can for instance be used together with the Lua function os.date(format, time) as the time variable.

Parameters: -

Returns:
Local time in seconds

Example:
```
var = xxter.localtime()
```

---

xxter.getmeteostatus()

Description:
Gives the amount of hours since the weather information was last updated. The weather information is updated automaticaly, but this function can be used to know when this was last done.

Parameters: -

Returns:
Number of hours since last weather update

Example:
```
var = xxter. getmeteostatus()
```

---

xxter.getmeteoinfo(what [, offset [, unit]])

Description:
This functions returns the value of the requested property. In case the data is not available, *false* is returned.

Parameters:
what : Property that is to be returned, see table 1 below (string)
offset : Offset in hours, for how far in the future the information should be (number)
For example “4” provides the weather information expected in 4 hours. Default is 0.
unit : Unit of the requested value (enum; “C”=Celsius, “K”=Kelvin, “F”=Fahrenheit)
Only applies to temperatures, default is “C”.

Returns:
Expected value of the provided property in the provided time period

Examples:
```
var = xxter.getmeteoinfo("temperature", 8, "F")
var = xxter.getmeteoinfo("rain", 24)
```

Example 1 gives the expected temperature in Fahrenheit in 8 hours.
Example 2 gives the rainfall that is expected in 24 hours.

Table 1: Possible properties for meteo functions		
“temperature”	Temperature (in C, F or K)	
"feels like”	Temperature by human perception (in C, F or K)	
“pressure”	Atmospheric pressure at sea level (in hPa)	
“humidity”	Relative air humidity (in %)	
"dew point”	Temperature at which water droplets begin to condensate (in C, F or K)	
"uv index”	Ultra violet radiation risk index	
“clouds”	Cloud coverage (in %)	
“visibility”	Average visibility (in meters)	**- n/a with min/max functions**
"wind speed”	Wind speed (in m/s)	
"wind gust”	Wind gusts, where available (in m/s)	
"wind degrees”	Wind direction, 0 = N, 90 = E, 180 = S, 270 = W	**- n/a with min/max functions**
“rain”	Expected amount of rain (in mm)	**- n/a with min function**
“snow”	Expected amount of snow (in mm)	**- n/a with min function**
“precipitation”	Expected amount of precipitation (in mm)	**- n/a with min function**
"precipitation chance”	Chance on precipitation (rain or snow)	**- n/a with min function**

xxter.getmeteoinfomin(what, from_offset, to_offset [, unit])

Description:
This functions returns the minimal value of the requested property over the provided period of time. In case the data is not available, *false* is returned.

Parameters:
what : Property that is to be returned, see table 1 on the previous page (string)
from_offset : Time in hours, from when the forecast should be (number)
to_offset : Time in hours, until when the forecast should be (number)
unit : Unit of the requested value (enum; “C”=Celsius, “K”=Kelvin, “F”=Fahrenheit)
Only applies to temperatures, default is “C”.

Returns:
Expected minimal value of the provided property over the provided time period

Examples:
```
var = xxter.getmeteoinfomin("temperature", 0, 8, "F")
var = xxter.getmeteoinfomin("clouds", 24, 48)
```

Example 1 gives the minimal temperature in Fahrenheit between now and the coming 8 hours.
Example 2 gives the minimal expected cloud coverage between 24 and 48 hours from now.

---

xxter.getmeteoinfomax(what, from_offset, to_offset [, unit])

Description:
This functions returns the maximum value of the requested property over the provided period of time. In case the data is not available, *false* is returned.

Parameters:
what : Property that is to be returned, see table 1 on the previous page (string)
from_offset : Time in hours, from when the forecast should be (number)
to_offset : Time in hours, until when the forecast should be (number)
unit : Unit of the requested value (enum; “C”=Celsius, “K”=Kelvin, “F”=Fahrenheit)
Only applies to temperatures, default is “C”.

Returns:
Expected maximum value of the provided property over the provided time period

Examples:
```
var = xxter.getmeteoinfomax("temperature", 0, 8, "F")
var = xxter.getmeteoinfomax("rain", 24, 48)
```

Example 1 gives the maximum temperature in Fahrenheit between now and the coming 8 hours.
Example 2 gives the maximum expected rain between 24 and 48 hours from now.

---

xxter.gettariff(delta_min)

Description:
Gets the actual tariff for the provided moment in time in the future (in minutes), as set for the Smart Energy Manager. If a variable tariff is configured this will be a dynamic value, otherwise it will return the single or split tariff applicable for that time.

Parameters:
delta_min : The amount of minutes in the future (number)

Returns:
The actual tariff, as set for the Smart Energy Manager.

Example:
```
var = xxter.gettariff(60)
```

---

xxter.getusername()

Description:
Gets the username of the user that started the function. When a (local) user triggers the script, for instance through an action or the app, his or her username will be returned with this function. If the script is triggered through any other means, for instance from a value in the automation, the main user will be returned.

Parameters: -

Returns:
The name of the user that started the function.

Example:
```
var = xxter.getusername()
```

Supported general Lua functions

Apart from these specific xxter functions, the following standard Lua functions are also supported:

Table 2: supported standard Lua functions

```
tonumber (e [, base])
tostring (v)
type (v)
coroutine.create (f)
coroutine.isyieldable ()
coroutine.resume (co [, val1, ···])
coroutine.running ()
coroutine.status (co)
coroutine.wrap (f)
coroutine.yield (···)
string.byte (s [, i [, j]])
string.char (···)
string.dump (function [, strip])
string.find (s, pattern [, init
                        [, plain]])
string.format (formatstring, ···)
string.gmatch (s, pattern)
string.gsub (s, pattern, repl [, n])
string.len (s)
string.lower (s)
string.match (s, pattern [, init])
string.pack (fmt, v1, v2, ···)
string.packsize (fmt)
string.rep (s, n [, sep])
string.reverse (s)
string.sub (s, i [, j])
string.unpack (fmt, s [, pos])
string.upper (s)
utf8.char (···)
utf8.charpattern
utf8.codes (s)
utf8.codepoint (s [, i [, j]])
utf8.len (s [, i [, j]])
utf8.offset (s, n [, i])
os.clock ()
os.date ([format [, time]])
```

```
os.difftime (t2, t1)
os.time ([table])
table.concat (list [, sep [, i [, j]]])
table.insert (list, [pos,] value)
table.move (a1, f, e, t [,a2])
table.pack (···)
table.remove (list [, pos])
table.sort (list [, comp])
table.unpack (list [, i [, j]])
math.abs (x)
math.acos (x)
math.asin (x)
math.atan (y [, x])
math.ceil (x)
math.cos (x)
math.deg (x)
math.exp (x)
math.floor (x)
math.fmod (x, y)
math.huge
math.log (x [, base])
math.max (x, ···)
math.maxnumber
math.min (x, ···)
math.minnumber
math.modf (x)
math.pi
math.rad (x)
math.random ([m [, n]])
math.randomseed (x)
math.sin (x)
math.sqrt (x)
math.tan (x)
math.tonumber (x)
math.type (x)
math.ult (m, n)
```

Documentation for these functions can be found in the Lua manual:
https://www.Lua.org/manual/5.3/