# EventRunner

[EventRunner 1](#_Toc72299889)

[Introduction 2](#_Toc72299890)

[Setting up EventRunner 3](#_Toc72299891)

[Installing EventRunner 3](#_Toc72299892)

[Configuring EventRunner 3](#_Toc72299893)

[EventScript 3](#_Toc72299894)

[Time constants 3](#_Toc72299895)

[Arithmetic operators 3](#_Toc72299896)

[Comparison operators 4](#_Toc72299897)

[Logical operators 4](#_Toc72299898)

[Date and day related functions 4](#_Toc72299899)

[EventScript variables 5](#_Toc72299900)

[EventScript functions 5](#_Toc72299901)

[Table operations 5](#_Toc72299902)

[Mathematical functions 5](#_Toc72299903)

[Type and name functions 6](#_Toc72299904)

[Log and formatting 6](#_Toc72299905)

[Rule and event functions 6](#_Toc72299906)

[Managing fibaro globals 6](#_Toc72299907)

[VD functions (only HC2) 6](#_Toc72299908)

[EventScript  functions 6](#_Toc72299909)

[Control structures 7](#_Toc72299910)

[Rule constructs 7](#_Toc72299911)

[Integration with other systems 8](#_Toc72299912)

[Property functions: 8](#_Toc72299913)

[Alarm handling 10](#_Toc72299914)

[EventScript syntax 10](#_Toc72299915)

[Events 11](#_Toc72299916)

[Events and Lua 11](#_Toc72299917)

[Recipes 11](#_Toc72299918)

[Scheduling actions at times 11](#_Toc72299919)

[Trigger rules 17](#_Toc72299920)

[Implementation notes 23](#_Toc72299921)

## Introduction

EventRunner is an automation framework for HC3 that aims to make it write home automation rules in a declarative style: IF <conditions> then <actions>

Lets start with a simple rule:

rule(“50:breached => 60:on”)

This will turn on the light with device id 60 when the sensor with device id 50 is breached.

The format for rules is

“<condition> => <actions>”

Rules are Lua strings that we give to the function rule(<string>) that compiles the rule to an internal representation. The rule is then recalled when the condition specified are true and the actions are executed. This is as simple as it is.

## Setting up EventRunner

### Installing EventRunner

### Configuring EventRunner

## EventScript

#### Time constants

Time constants are written in the format HH:MM or HH:MM:SS

Ex. 10:00 is ten o’clock. 10:00:30 is 30s after ten o’clock. If you only want to specify seconds you still need to have hours, ex. 00:00:30.

Time constants appearing in the rules are converted to the number seconds.  
This allows for arithmetic like

rule("11:20 == 10:30+00:50")

* HH:MM[:SS], 10:20 == 3600\*10+60\*20
* +/HH:MM[:SS],  Add time to current time. +/10:20 == os.time()+3600\*10+60\*20
* t/HH:MM[:SS],  Epoc of time today. t/10:20 == midnight+3600\*10+60\*20
* n/HH:MM[:SS], Epoc of next 10:20. Today if it's before 10:20 and 10:20 tomorrow if it's after 10:20

Time constants are normally used for time-of-day specifications as it only covers 24 hours. We call them short form time constants.

Internally the HC3 use a long form, epoch, that is the number of seconds since January 1, 1973.

In Lua os.time() returns the current time in epoch as a number.

* YYYY/MM/DD/HH:MM[:SS], Long form for Epoc. Ex. 2020/03/01/10:20

#### Arithmetic operators

* %,\*,+,-,/ - Normal operator precedence is used and parentheses is also allowed

rule("-7 + 8/2 == -3")  
>true

* '+=','-=','\*=' - increment and assign. works with variables only

rule("a = 7; a += 10; a == 17")  
rule("a = 7; a -= 10; a == -3")  
rule("a = 7; a \*= 10; a == 70")

#### Comparison operators

* <,<=,==,>,>=,~=

Rule.eval("6 > 4 & 'A' ~= 'B'")  
>true

#### Logical operators

* &,|,!

rule("6 > 5 & !(7 > 8) | true)  
>true

#### Date and day related functions

* HM(<time), converts a time to string  "HH:MM"

rule("log('Time is %s',HM(now))")  
>Time is 10:45

* HMS(<time), converts a time to string  "HH:MM:SS"

rule("log('Time is %s',HMS(now))")  
>Time is 10:45:10

* now, seconds since midnight

rule("sensor:isOn & now < sunset => log('Sensor triggered before sunset')")

* midnight, epoc for last midnight

rule("log('Time is %s',osdate('%H:%M',midnight+now))")  
>Time is 10:45

* dawn, seconds from midnight to dawn
* dusk, seconds from midnight to dusk
* sunrise, seconds from midnight to sunrise
* sunset, seconds from midnight to sunset
* wnum, week number
* date(<string>), cron test
* day(<string>)

rule("@sunset & day('1') => log('Sunset first day of the month')")  
rule("@sunset & day('1,2') => log('Sunset first two days of the month')")  
rule("@sunset & day('1-10') => log('Sunset first 10 days of the month')")  
rule("@sunset & day('last') => log('Sunset last day of the month')")  
rule("@sunset & day('lastw') => log('Sunset first day of last week of month')")  
rule("@sunset & day('lastw-last') => log('Sunset last week of month')")

* month(<string>)
* wday(<string>)
* osdate(<string>)
* ostime(), current time in epoc (midnight+now)
* time(<string>), converts time string to seconds

### EventScript variables

### EventScript functions

#### Table operations

* add(<table>,<element>)
* remove(<table>,<index>)
* size(<table>)

#### Mathematical functions

* rnd(<n1>,<n2>)
* round(<number>)
* sign(<number>)
* average(<table>)
* max(<table>)
* min(<table>)
* sum(<table>)
* sort(<table>)

#### Type and name functions

* num(<element>)
* idname(<deviceID>)
* str(<element>)
* type(<element>)

#### Log and formatting

* log(<string>,...)
* fmt(<string>,...)
* fjson(<string>)
* tjson(<table>)

#### Rule and event functions

* post(<event>,<time>)
* cancel(<post ref>)
* remote(<sceneID>,<event>)
* publish(<event>)
* subscribe(<event pattern>)

#### Managing fibaro globals

* global(<string>)
* listglobals()
* deleteglobal(<string>)

#### VD functions (only HC2)

* label(<deviceID>,<tag>)
* label(<deviceID>,<tag>)=<string>
* slider(<deviceID>,<tag>)
* slider(<deviceID>,<tag>)=<string>
* VDev.define(<name>,<tag>,<version>,<rows>)
* VDev.remove(<tag>)
* VDev.clearCache()

#### EventScript  functions

* env - variable that contains the event that has triggered the rule

rule("88:isOn => log('Event:%s',env.event)")  
-- HC2  
>{type='property', deviceID=88, propertyName='value', value=1}  
-- HC3  
>{type='device', id=88, properyty='value', value=true}

* eval(<string>)
* disable(<rule>)
* enable(<rule>)
* trueFor(<time>,<expression>)
* again(<n>)
* once(<expression>) -- Expression needs to turn false before it will be true again.

rule("once(sensor:temp > 10) => log('power is over 10')")

The rule will rigger the first time the temp > 10, then it needs to go below 10 and above again for it to trigger the next time.

rule("sensor:breached & once(06:00..07:00) => sonos:play='Good morning'")

Plays good morning message only once when sensor breached between 6-7

* wait(<time>)
* trace(<boolean>)
* match
* pack(<function call>)

#### Control structures

* if <expr> then <statments> elseif <expr> then <statments> else <statments> end
* while <expr> do <statments> end
* repeat <statments> until <expr>
* for i=<expr>,<expr>[,<expr>] do <statments> end
* for k,v in ipairs(<table>) do <statments> end
* for k,v in pairs(<table>) do <statments> end
* || <test> >> <statments> ...

#### Rule constructs

* @<time> [& <additional tests>] => <statements>  
  Schedule rule at <time>  every day
* @@<time> [& <additional tests>] => <statements>  
  Schedule rule at repeating <time> intervals
* <trigger expression> => <statements>  
  <trigger expression> contains triggers (deviceID, fibaro globals), and rule will be called every time they change status
* #<event> [& <additional tests>] => <statements>  
  Rule will be called every time #<event> is posted

#### Integration with other systems

* Nodered.connect(<url>)
* Nodered.post(<event>)
* Hue.connect(<user>,<ip>) (only HC2)
* Hue.define(<name>,<deviceID>) (only HC2)
* Hue.monitor(<deviceID>,<ms>) (only HC2)
* Hue.dump() (only HC2)
* Telegram.bot(<token>) (only HC2)
* Telegram.msg(<id>,<string>) (only HC2)

#### Property functions:

* <ID>:value - returns value property of device. fibaro.getValue(id,"value")
* <ID>:value=..  - set value property of device. fibaro.call(id,"setValue",...)
* <ID>:isOff. -- returns true if device value is false or 0
* <ID>:isOn -- returns true if device value is true or > 0
* <ID>:isAllOn
* <ID>:isAnyOff
* <ID>:off -- turns off device. fibaro.call(id,"turnOff")
* <ID>:on - turns on device. fibaro.call(id,"turnOn")
* <ID>:toggle
* <ID>:dim={<time>[,'up'|'down',<step>,<curve>,<start,<stop>}
* <ID>:breached,   synonym for :isOn but more appropriate for motion/door/window sensors
* <ID>:safe,  synonym for :isOff but more appropriate for motion/door/window sensors
* <ID>:isClosed
* <ID>:isOpen
* <ID>:close
* <ID>:open
* <ID>:stop
* <ID>:bat
* <ID>:power
* <ID>:power=..
* <ID>:lux,  synonym for :value but more appropriate for lux sensors
* <ID>:temp,  synonym for :value but more appropriate for temperature sensors
* <ID>:last
* <ID>:trigger
* <ID>:secure
* <ID>:unsecure
* <ID>:lock, synonym for :secure
* <ID>:unlock, synonym for :unsecure
* <ID>:isSecure
* <ID>:isUnsecure
* <ID>:isLocked, synonym for :isSecure
* <ID>:isUnlocked, synonym for :isUnsecure
* <ID>:manual
* <ID>:wake
* <ID>:name
* <ID>:HTname
* <ID>:roomName
* <ID>:start
* <ID>:start=..
* <ID>:kill
* <ID>:scene
* <ID>:access
* <ID>:central
* <ID>:armed, fibaro:getValue(<ID>,"armed")=='1'
* <ID>:armed=<expr>, fibaro:call(<ID>,"setArmed",<expr> and '1' or '0')
* <ID>:btn=..
* <ID>:dID
* <ID>:defemail=..
* <ID>:email=..
* <ID>:msg=..
* <ID>:defaultPartyTime=..
* <ID>:removeSchedule
* <ID>:retryScheduleSynchronization
* <ID>:interval=..
* <ID>:schedule=..
* <ID>:scheduleState=..
* <ID>:setAllSchedules
* <ID>:setpointMode=..
* <ID>:targetLevel=..
* <ID>:thermostatSetpoint=..
* <ID>:mode
* <ID>:mode=..
* <ID>:time
* <ID>:time=..
* <ID>:B=<expr>, fibaro:call(<ID>,"setB",<expr>)
* <ID>:G=<expr>, fibaro:call(<ID>,"setG",<expr>)
* <ID>:R=<expr>, fibaro:call(<ID>,"setR",<expr>)
* <ID>:W=<expr>, fibaro:call(<ID>,"setW",<expr>)
* <ID>:B, fibaro:getValue(<ID>,"B")
* <ID>:G, fibaro:getValue(<ID>,"G")
* <ID>:R, fibaro:getValue(<ID>,"R")
* <ID>:W, fibaro:getValue(<ID>,"W")
* <ID>:color
* <ID>:color=..

…

#### Alarm handling

**Arming**  
rule("2:alarm=true")  -- arm partition 2  
rule("0:alarm=true")  -- arm all partitions  
**Disarming**  
rule("2:alarm=false")  -- disarm partition 2  
rule("0:alarm=false")  -- disarm all partitions  
**Check/trigger if armed**  
rule("2:armed => log('partition 2 armed')")  
rule("0:armed => log('house armed')")

rule("{...}:armed => log('at least one is armed')")  
rule("{...}:allArmed => log('all armed')")  
rule("#alarm{id='$id',property='armed',value=true} => log('partition %s armed',id)")  
rule("#alarm{property='homeArmed', value=true} => log('house armed')")  
**Check/trigger if disarmed**  
rule("2:disarmed => log('partition 2 disarmed')")  
rule("0:disarmed => log('house disarmed')")

rule("{...}:disarmed => log('all are disarmed')")  
rule("{...}:anyDisarmed => log('at least one is disarmed')")  
rule("#alarm{id='$id',property='armed',value=false} => log('partition %s disarmed',id)")  
rule("#alarm{property='homeArmed', value=false} => log('house disarmed')")

**Watch/trigger if partition is enabled (starting 'exit delay')**

rule("2:alarm='watch'; log('watching partition 2')")  
rule("0:alarm='watch'; log('watching all partitions')")

rule("2:willArm => log('partition 2 will arm in %ss',env.event.value)")

rule("0:willArm => log('house will arm soon')")

rule("#alarm{id='$id',property='willArm',value='$secs'} =>  log('partition %s will arm in %ss',id,sec)")

**Getting name of partition**

rule("log('partition 2 name is %s',2:alarm.name)")

**Check if a partition have breached sensors**

rule("log('partition2 has %s devices breached',2:alarm.devices:breached & 'some' | 'no')")

### EventScript syntax

<symbol> ::= (a-zA-Z) (a-zA-Z\_0-9)

<global> ::= $<symbol>

<var> ::= <symbol>

<event> ::= #<symbol>{<symbol>=<expr>,...,<symbol>=<expr>}

<constant> ::= 'env' | 'now' | 'midnight' | 'sunrise' | 'sunset' | 'dusk' | 'dawn' | 'wnum' | 'true' | 'false' | 'nil' | '{}'

<time> ::= HH:MM | HH:MM:SS | YYYY/mm/DD/HH:MM | YYYY/mm/DD/HH:MM:SS

<timeExpr> ::= <time> | +/<time> | n/<time> | t/<time>

<oper> ::= + | - | \* | / | % | "|" | & | > | < | >= | <= | == | = | =~ | = | .. | .| : | ! | @

<expr> ::= <table> | <call> | <event> | <num> | <string> | <constant> | <var> | <global> | <timeExpr> | <addr>

<expr> ::= <expr> <oper> <expr> | - <expr> | ( <expr> )

<expr> ::= fn(<var>,...,<var>) <statements> end

<expr> ::= <expr> ? <expr> : <expr> -- not implemented yet...

<call> ::= <fun>(<expr>, ..., <expr>) | <var>(<expr>, ..., <expr>) | (<expr>)(<expr>, ..., <expr>)

<statements> ::= statement [ ; <statements> ]

<statement> ::= <expr>

<statement> ::= || <expr> >> <statements> {|| <expr> >> <statements>} [;;]

<statement> ::= while <expr> do <statements> end

<statement> ::= repeat <statements> util <expr> end

<statement> ::= for <var> = <expr>,<expr>[,<expr>] do <statements> end

<statement> ::= for <var>[,<var>] in {ipars | pairs} do <statements> end

<statement> ::= if <expr> then <statements> {elseif <expr> then <statements>} [else <statements>] end

<rule> ::= @<time> [& <expr>] => <statements>

<rule> ::= @{<time>, ..., <time>} [& <expr>} => <statements>

<rule> ::= @@<time> [& <expr>] => <statements>

<rule> ::= <expr> => <statements>

<rule> ::= <event> => <statements>

## Events

### Events and Lua

## Recipes

### Scheduling actions at times

The full blown [EventRunner](https://github.com/jangabrielsson/EventRunner) framework has support for more advanced event programming than the EventRunnerLite described in the first post of this tutorial. In this post the script language implemented in the EventRunner framework will be described. The script language is particularly suitable for writing compact rules handling time scheduling, device triggering of rules, and user defined event rules.

This first part will describe how script rules can be used to define a flexible scheduler. A scheduler is a scene that runs actions at specific times during the day. A flexible scheduler can adjust its behaviour depending on day of week, month or state of global variables or state of other devices at that time.

In the EventRunner framework, the 'main()' function is used to setup rules and is called before the framework starts. 'main()' can also be used to read in a HomeTable, setup variables etc, things that needs to be done once the framework starts up.

To define a rule the 'Rule.eval(<string>)' function is used. A rule is a Lua string of the format "<condition> => <actions>", and is compiled to an efficient representation that is run when the <condition> is true. If the string doesn't contain a '=>' it is considered to be an expression that is just evaluated, and the result is returned. This can be useful to setup variables and other initializations.

function main()

Rule.eval("lamp=55") -- define variable 'lamp' to be 55

Rule.eval("lamp:on") -- turn on lamp (with deviceID 55) once when scene starts

End

A scheduler runs actions at a specific time during the day, so we use the 'daily’ or '@' rules

The generic form of a daily rule is "@<time> [<optional extra tests>] => <actions>"

function main()

Rule.eval("lamp=55") -- define variable 'lamp' to be 55

Rule.eval("@15:10 => lamp:on") -- turn on lamp (deviceID 55) every day at 15:10

end

The rule will be called every day at 15:10, and the left-hand expression "@15:10" will evaluate to true (because it's 15:10) and thus the right-hand side if the '=>' will be carried out. In this case the lamp with deviceID 55 will be turned on.

However, this also means that we can tuck on extra tests on the left-hand side that needs to be true for the rule to execute its actions. Any logic expression combining AND (&), OR(|), NOT(!), or comparison operators (==,>=,<=,~=) can be used.

Ex. (we don't include the 'function main()' part in the examples from now on)

Rule.eval("@15:10 & lamp:isOff => lamp:on") -- turn on lamp (deviceID 55) every day at 15:10, if the lamp is off.

Rule.eval("@15:10 & !lamp:isOn => lamp:on") -- equivalent to the previous rule.

This tests if the lamp also is off, and if so the lamp is turned on. It is important that there should be no "side-effects" on the left-hand side. No functions that turn on or off devices or set globals etc, only functions that query states.

Here are some examples of types of rules that can be defined

 -- Declare a script variable 'lamp' to have value 55 (e.g. a deviceID)

 Rule.eval("lamp=55")

  -- Every day at 07:15, turn of lamp, e.g. deviceID 55

 Rule.eval("@07:15 => lamp:off")

  -- Every day at sunrise, turn off lamp

 Rule.eval("@sunrise => lamp:off")

  -- Every day at sunrise + 15min, turn off the lamp

 Rule.eval("@sunrise+00:15 => lamp:off")

  -- Every day at sunset, turn on lamp

 Rule.eval("@sunset => lamp:on")

  -- Every day at sunset-15min, if lamp is off, turn on the lamp

 Rule.eval("@sunset-00:15 & lamp:isOff => lamp:on")

  -- Every day at sunset and if it's Monday, turn off the lamp

 Rule.eval("@sunrise & wday('mon') => lamp:off")

  -- Every day at sunrise and if it is a weekday, turn off the lamp

 Rule.eval("@sunrise & wday('mon-fri') => lamp:off")

  -- Every day at sunrise on Monday,Wednesday,Friday,Saturday,Sunday, turn off the lamp

 Rule.eval("@sunrise & wday('mon,wed,fri-sun') => lamp:off")

 -- Every day at sunrise the first day of the month, turn off the lamp

 Rule.eval("@sunrise & day('1') => lamp:off")

 -- Every day at sunrise on the first 15 days of the month, turn off the lamp

 Rule.eval("@sunrise & day('1-15') => lamp:off")

 -- Every day at sunrise on the last day of the month, turn off the lamp

 Rule.eval("@sunrise & day('last') => lamp:off") -- 'last' is the number of the last day in the current month

 -- Every day at sunrise on the first day of the last week of the month, turn off the lamp

 Rule.eval("@sunrise & day('lastw') => lamp:off") -- 'lastw' is the number of the first day in the last week of the current month

 -- Every day at sunrise on a Monday in the last week of the month, turn off the lamp

 Rule.eval("@sunrise & day('lastw-last') & wday('mon') => lamp:off")

 -- Every day at sunrise January to Mars, turn off the lamp

 Rule.eval("@sunrise & month('jan-mar') => lamp:off")

 -- Every day at sunrise on Mondays at even week numbers, turn off the lamp

  Rule.eval("@sunrise & wnum%2 == 0 & wday('mon') => lamp:off")

 -- Every day at sunrise on weekdays when fibaro global 'Presence' equals 'Home', turn off the lamp

 Rule.eval("@sunrise & $Presence=='Home' & wday('mon-fri') => lamp:off")

-- Every day at sunrise on weekdays when fibaro global 'Presence' equals 'Home' or fibaro global 'Simulate' equals 'true', turn off the lamp

 Rule.eval("@sunrise & ($Presence=='Home' | $Simulate='true') & wday('mon-fri') => lamp:off")

The above rules run at sunrise every day and we add additional conditions that restrict it to sunrise at specific days and/or if a global variable also is set to a specific value or if a device is in a specific state.

All 'daily/@' rules are 'examined' at midnight and the expression after the '@' character is computed and should return a number being the seconds after midnight the rule should be run. That is why there should be no side-effects in the left-hand side of the rule as they would be carried out at midnight, which we probably don't want. When the rule is later run at the computed/specified time, the whole left-hand side is computed again as a logical expression and if it returns true the right-hand side, the action(s), is run.

'sunrise' and 'sunset' are constants that return seconds to sunrise and sunset respectively. Because the '@' expression is computed we can specify expressions like 'sunset+00:15' and it's computed as 15 min after sunset in seconds. Time constants like '04:15' are shorthand for '0+(60\*(15+60\*4))'. Seconds can also be specified '10:20:30' same as '30+(60\*(20+60\*10))'

We can use a value from a global variable easily, but if we want to use the time notation we need to convert it to seconds because globals always return strings. The 'time' function converts a string with a time value to seconds.

Rule.eval("$Morning = '07:00'") -- Set global $Morning to the string "07:00". Could be set by a VD instead

Rule.eval("@time($Morning) => log('It's morning!')")

or if we want to adjust and offset to 'sunrise' with a global variable, maybe controlled from a VD

Rule.eval("@sunrise+time($SunriseOffset) => log('It's morning')")

 Remember that the times are calculated at midnight, so if the global is changed during the day it will not take effect until the next day (However, the scene can be restarted for all values to be re-calculated).

A typical case is to turn on a light in the morning if that time is before sunrise

Rule.eval("@06:00 & now < sunrise+00:30 => lamp:on")

 This rule is run 06:00 every morning but the constant 'now', representing the current number of seconds since midnight, must be less than sunrise + 30min for the right-hand action to be run.

Maybe a lamp should be turned on in the afternoon at sunset, given that sunset is within a certain time window (here in the north we can have sunset at 2PM)

 Rule.eval("@sunset-00:30 & 16:00..23:00 => lamp:on")

 The '<time1>..<time2>' operator is true if the current time is between the specified times.

To do something at a random interval every day, the 'rnd' function can be used. 'rnd(x,y)' returns a random number between x and y.

Rule.eval("@sunset+rnd(-00:30,00:30) => lamp:on") -- Turn on lamp at sunset +/- 30min

 Assume we have different wake-up times depending on day of week. A short alarm clock could look like this.

Rule.eval("phone=109") -- ID of phone

Rule.eval("wakeUpTime={Mon=07:10,Tue=07:20,Wed=06:55,Thu=07:40,Fri=07:30,Sat=08:00,Sun=09:00}")

Rule.eval("@wakeUpTime[osdate('%a')] & $Presence~='Vacation' => phone:send=log('Time to wakeup')")

 This will at every midnight schedule the wakeup time associated with the weekday that we get from 'osdate' that takes same arguments as Lua's 'os.date'.

'<ID>:send=<string>' send a text string to a phone with id ID. The 'log' command prints its message to the HC2 debug window but also returns the string which we then use as input to the ':send' command. We also make sure that our global variable 'Presence' is not set to 'Vacation', as we don't want any messages then.

The '@' expression can also be applied to a list of times, and all will be scheduled.

If we want to do something at every 15min between 10:00 and 14:00, it's easiest to do something like this

Rule.eval("flowerCheck = {10:00,10:15,10:30,10:45,11:00,11:15,11:30,11:45,12:00,12:15,12:30,12:45,13:00,13:15,13:30,13:45,14:00}")

Rule.eval("@flowerCheck => wday('mon-fri') & checkWater()") -- assumes we have a defined function 'checkWater()'

The advantage is that this is done every 15min between 10:00 and 14:00 every weekday and there is no time drift.

There is another construct that schedule actions at specific intervals, the '@@' operator.

Rule.eval("@@00:15 => log('Dong!')") -- logs 'Dong!' every 15min 24x7...

'@@' is drift free, so it will execute exactly on the interval specified, starting at what ever time it starts.

Here is a really short presence simulator

Rule.eval("lamps={22,33,44,55,66,77,88}")

Rule.eval("$Presence=='away' => lampStates = lamps:value") -- Save lamps current states

Rule.eval("$Presence=='home' => lamps:value = lampStates") -- Restore lamp states

Rule.eval("@@rnd(00:10,00:30) & $Presence=='away' & sunset..sunrise => lamps[rnd(1,length(lamps))]:toggle")

Rule.eval("wait(00:10); $Presence='away'; wait(01:00); $Presence='home'") -- Test rule by triggering global 'Presence'

 This runs at random intervals between 10 and 30min turning on/off lamps, when the global variable 'Presence' is set to 'away' and it's between sunset and sunrise. We select a random lamp from the 'lamps' table and call the ':toggle' function to toggle the state of the lamp.

When the 'Presence' global is set we save and restore the lamp values (triggering rules will be explained in a separate post)

There is another way to achieve things being scheduled at specific intervals.

Rule.eval("@@00:01 & date('0 10-16/2') => log('Hupp')")

It is possible to use the date(<crontab format string>) command that is almost a crontab test  (same format and it compiles to a code that test the condition very efficient).

date('0 10-16/2') means on the '0' minute every second hour from 10 to 16.

To make something on every hour, on the hour, the string is simply date('0 \*') - \* stands for any hour, the same as date('0 0-23'). Much simpler than having to list all hours with the daily '@' command.

Crontab is very flexible, date('15,45 7-19/3 \* dec,jan mon-fri') means 15min and 45min past every third hour between 7 and 17 on Monday to Friday in January and December.

The '@@00:01' will run the test every minute (exactly like crontab does) and if the date() test is true it will run the action. This is very flexible.

The day(),wday(),month() functions are implemented with date(). Have a look at any crontab documentation on the net and you will get the gist of it.

One more note. In the examples above we have declared a local script variable ‘lamp’ that we use in the actions. It is easy to bring in HomeTable definitions to be used in the scripts.

Assume there is a hometable that looks like this:

jT = {kitchen={lamp=55,sensor=88},bedroom={lamp=57,sensor=89}}

The function ‘Util.defvars’ will declare the table as script variables.

Util.defvars(jT)

Rule.eval(“@sunset => kitchen.lamp:on; bedroom.lamp:on”)

So, script rules allow us to write compact and flexible rules for scheduling actions. It's easy to integrate with a VD and fibaro globals to adjust rules for specific contexts. The syntax for the script language and available action functions is described in more detail here <[link](https://github.com/jangabrielsson/EventRunner/wiki/Script-expressions)>

 Next up is a post on trigger rules...

### Trigger rules

--[[

%% properties

55 value

66 value

77 value

78 sceneActivation

88 ui.Slider1.value

88 ui.Label1.value

%% events

100 CentralSceneEvent

120 AccessControlEvent

%% globals

Home

%% autostart

--]]

myLightSensor = 55 -- do not declare local, script will not find them(!)

myLight1 = 66

myLight2 = 67

myDoorSensor = 77

mySwitch = 78

myVD=88

myKeyFob = 100

myLock = 120

function main()

 Rule.eval("myLightSensor:lux > 200 => myLight1:on")       -- Turn on light1 if lux value goes above 200

 Rule.eval("myLight1:isOn => myLight2:on")                 -- Turn on light2 if light1 is turned on

 Rule.eval("myDoorSensor:breached => myLight1:on")         -- Turn on light1 if door sensor is breached

 Rule.eval("mySwitch:scene == S2.click => myLight1:on")    -- Turn on light1 if S2 is clicked once

 Rule.eval("slider(myVD,'Slider1') == 50 => myLight1:on")  -- Turn on light1 if slider is set to 50

 Rule.eval("label(myVD,'Label1') == 'ON' => myLight1:on")  -- Turn on light1 if label is set to 'ON'

 Rule.eval("myKeyFob:central.keyId==4 => myLight1:on")     -- Turn on light1 if key 4 is pressed on keyFob

Rule.eval("myLock:access.status=='Unlock' => log('Door unlocked by %s',myLock:access.name)") -- Door unlocked

 Rule.eval("$Home == 'AWAY' => myLight1:on")               -- Turn on light1 if fibaro global variable 'Home' is set to 'AWAY'

Rule.eval("#AccessControlEvent{data={name='$name',slotId='$slot',status='Unlock',id=myLock}} => log('Door unlocked by %s',name)")

end

This is pretty simple examples. Events need to be declared in the scene header (or the EventRunner scene will never be informed about the events). After that, rules reacting on events can be written straight forward in the script syntax supported.

A trigger rule has the syntax

<expression involving scene triggers> => <actions>

When the script is "compiled" we look through the left-hand expression looking for any device/global or other event and register the rule to be called whenever those devices/globals or other events arrive.

So when 'myLightSensor's value changes in the above example, the scene is triggered and the EventRunner framework make sure to call the right rule(s). The left-hand side is then evaluated, 'myLightSensor:lux > 200', and if it returns true the right-hand side is evaluated which turns on 'myLight1'.

The last rule is an event 'AccessControlEvent' that the framework currently hasn't special syntax support for. However, we can still match against  the incoming event (sourceTrigger) and write our rule. In the future we may add script syntax to deal with these event more conveniently.

The left-hand side can contain expressions involving several triggers, and the rule is called whenever any changes state

Rule.eval("myDoorSensor:breached & $Home=='AWAY' => myLight1:on") -- Turn on light if sensor breached and 'Home' is 'AWAY'

This rule is called whenever ‘myDoorSensor’ changes state or fibaro global 'Home' change value. The left is evaluated so the sensor must be breached and the global set to 'AWAY' for the expression to be true and the light turned on.

This can be quite useful. Assume we have a set of fibaro motion and light sensors

Rule.eval("sensors={41,43,46,48}")

Rule.eval("phones={101,102}")

Rule.eval("sensors:breached => d=env.event.deviceID; phones:msg=log('Sensor %s in room %s breached',d:name,d:roomName')")

Rule.eval("max(sensors:lux) < 200 => myLight1:on") -- max value

Rule.eval("sum(sensors:lux)/size(sensors) < 200 => myLight1:on") -- average value

The first rule is called whenever a sensor changes state. ':breached' called on a set of devices will return true if any of the devices in the set is breached. The sourceTrigger that caused the rule to be triggered is available in the script variable 'env.event'. We pick out the deviceID that caused the trigger and assign it to a script variable 'd'. Then we apply the ':msg' operator on the set of phone IDs we want to send a message to. The message is created with the 'log()' function that writes the message to the log but also return the message string to the ':msg' operator so that it is sent to the phone devices. There are ':name' and ':roomName' operator we can use on a device to get the device name and room name.

The second rule is also called whenever a sensor changes state and but here we take out the max value from the set of lux values returned. If that max lux value is lower than 200 we turn on the light.

The third rule is similar but a more complex expression that sums the lux values and divide with the number of sensors to get an average lux value to check against.

In reality you may want to turn on and off the value depending on the lux value, but not toggling the light like crazy if the lux value is varying around the break point. Then the below can work.

Rule.eval("lux=sum(sensors:lux)/size(sensors) & math.abs(lux-200)>25 => lux < 200 & myLight1:on | myLight1:off")

The EventRunner framework calls rules immediately when an event comes in (it doesn't poll every x seconds) so it is suitable for writing rules that need to react quickly, like turning on lights when sensors are breached. Rules can also share local variables and states as they run in the same scene instance which further helps in writing rules.

Rule.eval("sensors:safe & door:safe => away=true")

Rule.eval("sensors:breached | door:breached => away=false")

Rule.eval("@@rnd(00:05,00:20) & away => lights[rnd(size(lights))]:toggle")

 Here, whenever all sensors are safe and also the door sensor is safe we set a local script variable 'away' to true.

The we have a schedule rule that runs at random intervals between 5 and 20min toggling lights, but only if away is true. A real presence simulation rule would not set 'away' until the sensors been safe for a period of time. For that it is possible to use the 'for' function described later in this post.

It is easy to limit rules to time intervals using the '..' operator

Rule.eval("sensor:breached & 06:00..08:00 => text2speech('God morning!')")

 Assuming we have a text2speech() function, if the sensor is breached and the time is between 06:00 and 08:00, the message plays.

This rule is called whenever the sensor changes state but also at 06:00 and 08:01. The reason is that the sensor can be breached at 05:59, which would make the left-hand side condition false and the action not invoked. However, at 06:00 the sensor is still breached but not changing state, so the rule would not be run. Therefore, rules containing '..' intervals are called at entry and exit of the intervals.

Of course, all kinds of tests can be added to a trigger rule, like day of week, month, week number etc. but they don't trigger the rule.

Rule("sensor:breached & 06:00..08:00  & wday('mon-fri') => text2speech('God morning!')")

This will trigger on sensor and time interval, but the wday test also needs to be true too for the action to be invoked.

In the example above, if the intention is to play the message once in the morning, the problem is that it will play every time the sensor is breached between 06:00 and 08:00. We could solve that with setting a flag first time the sensor is breached and clear it outside the intervall, However, it is easier with the 'once' function.

Rule("sensor:breached & once(06:00..08:00)  & wday('mon-fri') => text2speech('God morning!')")

'once' takes an expression and return true if the expression returns true. However, the expression needs to return false and then return true before 'once' will return true again. In this case it will return true the first time we are between 06:00 and 08:00 but then it will not be true until we exit the interval and enter it the next time, e.g. the next day. So, 'once' keeps state, similar to what we would have done with a flag and it would have been difficult to implement without rules running in the same scene instance.

Another useful function is 'for' that allow us to write tests that need to be true for a period of time.

Rule("sensor:breached & wday('mon-fri') => light1:on)")

Rule("trueFor(00:05,light1:isOn & sensor:safe) & wday('mon-fri') => light1:off)")

The 'for' function always return false the first time it is called, but if the expression is true it starts a timer for the time specified and if the expression is still true then it returns true and continue with the rule. If the expression turns false during the time the timer will be cleared.

The result is that we can easily write rules that check if something is true for a certain time, like windows left open etc.

Rule.eval("sensors={41,43,46,48}") -- window sensors

Rule.eval("phones={101,102}")

Rule.eval("trueFor(00:10,sensors:breached) & month('dec-feb') => d=env.event.deviceID; phones:msg=log('Window %s in room %s open',d:name,d:roomName')")

If any window is open for more than 10min a message is sent to the phones (during winter months)

Like 'once', 'for' will not re-trigger before the expression has turned false. In the above example, someone have to close all windows and then open one before the rule will trigger again. However, with 'repeat()' we can re-trigger the 'for' expression.

Rule.eval("trueFor(00:10,sensors:breached) & month('dec-feb') => d=env.event.deviceID; phones:msg=log('Window %s in room %s open',d:name,d:roomName'); repeat()")

This will make the rule trigger, and messages sent, every 10min the windows continue to be open. i.e. keep reminding us.

'repeat' can take an argument being the number of times the 'for' should be re-triggered. 'repeat' also return how many times it has currently repeated, e.g. re-triggered.

Rule.eval("trueFor(00:10,sensors:breached) => d=env.event.deviceID; phones:msg=log('Window %s in room %s open for %smin',d:name,d:roomName',10\*repeat(5))")

 This will remind us 5 times that the windows are open, '...open for 10min', to '...open for 50min',

Rules can also trigger on events.

Rule.eval("#property{deviceID=88,value='$>0'} => log('device 88 turned on!')")

This rule trigger on a {type='property', deviceID=88, propertyName='value'} event and in addition checks that the value filed is larger than 0. This is equivalent to write

Rule.eval("88:isOn => log('device 88 turned on!')")

...so for standard fibaro events it's seldom necessary. However it's useful for own defined events.

Rule.eval("sensor1:breached => post(#atHome)")

Rule.eval("door1:breached => post(#atHome)")

Rule.eval("#atHome => log('someone came home')")

Triggers for CentralSceneEvent or AccessControlEvent can be written using the ':central' and ':access' properties

 Rule.eval("myKeyFob:central.keyId==4 => myLight1:on")     -- Turn on light1 if key 4 is pressed on keyFob

Rule.eval("myLock:access.status=='Unlock' => log('Door unlocked by %s',myLock:access.name)") -- Door unlocked

Here, myKeyFob:central will trigger a CentralSceneEvent for the device with ID myKeyFob. ':central' will return the data part of the event so that we can continue to test if the result '.keyId' equals 4. If that's true we execute the right hand side and turn myLight1.

A problem is that a ':property' applied to a table of IDs will return a table of results. I.e. {66,77}:value is the same as {66:value, 77:value}, i.e. a table.

This is a problem because if we want to test if any of 2 keyFobs (id 101 and 103) has the '4' key pressed, we can't write

 Rule.eval("{101, 103}:central.keyId==4 => myLight1:on")

The reason is that it will return a table of keyId values for 101 and 103 and a table will not be equal to '4'. What we want to do is apply ':central' to the ID that triggered the rule.

':dID' will return the id that caused the event, so we can write like this

 Rule.eval("{101, 103}:dID:central.keyId==4 => myLight1:on")

':dID' returns the ID that triggered the event and then we apply ':central' on that to get the data part, and then we pick out the keyId key and can do our test.

Script rules comes in four main flavours, and always contain a '=>'

Rule.eval("@(<time>) & <expression> => <expression>")  -- daily schedules

Rule.eval("@@(<time>) & <expression> => <expression>") -- interval schedules

Rule.eval("<trigger expressions> => <expression>") -- trigger rules

Rule.eval("#<event> & <expression> => <expression>")   -- event rules

Rules are expressed as Lua strings and are compiled and registered with the function Rule.eval(<rule string>)

Later when matching conditions are met the rule is executed - conditions are on the left-hand side of the '=>' and actions on the right-hand side

Expression given to Rule.eval without an '=>' is evaluated immediatly.

Rule.eval("5+6")

>11

@time -- always part of a rule. Run action at time of day

Rule.eval("@10:00 => lamp:on") -- turn on lamp at 10:00 every day

@{t1,...tn} -- always part of a rule. Run action at times of day

Rule.eval("@{sunset+00:15, sunrise-00:20} => lamp:on") -- turn on lamp at sunrise minus 15min and sunset plus 15min every day

@{catch,t1,...} -- if the first argument in the time parameter list is the constant 'catch' the framework tries to catchup actions that have already passed when the framework starts up

Rule.eval("@{catch,10:00} -- If it's after 10:00 when starting up run the action anyway and schedule the next for next day..

@@intervall -- always part of a rule. repeat action at specified interval

Rule.eval("@@00:10 => lamp:on") -- turn on lamp every 10min Rule.eval("@@rnd(00:10,00:40) => lamp:on") -- turn on lamp random interval between 10min and 40min

#event -- event triggering rule.

'#foo' is a short for {type='foo'}

'#foo{bar=9}' is a short for {type='foo', bar=9}

Rule.eval("#property{deviceID=9,value='$>0'} => log('Device 9 turned on')") -- same as Rule.eval("9:isOn => log('Device ...)")

Matching with constraints are allowed in event values and start with '$..'. Variables bound in matching expression available as local vars in rest of rule.

Rule.eval("#global{name='Foo',value='$a'} => log('Fibaro global Foo's value set to %s',a)")

Rule.eval("#property{deviceID=78,value='$a>0'} => log('Dimmer 78 turned on with value %s',a)")

'env' -- script variable that is bound the the environment the script is executing in. Specifically env.event is the event (sourceTrigger) that triggered the rule.

Rule.eval("#foo => log('inifinte loop'); post(env.event)")

Rule.eval("#property{deviceID=66} => log('deviceID 66 value is %s',env.event.value)")

## Implementation notes

* Use styles to easily format your Word documents in no time. For example, this text uses the List Bullet Point style.
* In the Home tab of the ribbon, take a look at Styles to apply the formatting you want with just a tap.

|  |  |  |
| --- | --- | --- |
|  | Column Heading | Column Heading |
| Row Heading | Text | 123.45 |
| Row Heading | Text | 123.45 |
|  |  |  |