FIBARO System

Lua API Developer Documentation

Note:This documentation describes features available in Home Center version 4.0 Beta or newer.

Devices control

fibaro:call(deviceID, actionName, …)

Name

Function name must be always the same: **fibaro:call**

Application

Function sends a request to device to perform an action.

Parameters

deviceID: device ID

actionName: string containing action name

Additional (0 to 7) parameters may be inserted and sent as calling action arguments.

Returned values

None

Code example

```
-- non-parametric 'turnOff' action call of id=12 device
  fibaro:call(12, 'turnOff');
  -- 'setValue' action call with one additional parameter
  fibaro:call(12, 'setValue', '23');
```

fibaro:get(deviceID, propertyName)

Name

Function name must be always the same: **fibaro:get**

Application

Gets the latest data (value and time of last modification) of the device properties.

Parameters

deviceID: device ID number

propertyName: name of property

Returned values

string containing current property value

timestamp: last modification timestamp

Code example

```
-- get a value and time of the last "brightness" property modification (device id=11)
   local value, modificationTime = fibaro:get(11, 'brightness');
   -- second value may be omitted
   local value2 = fibaro:get(11, 'brightness');
   -- returned value may be used as a scene condition
   if (tonumber(value) >= 50) then
     fibaro:call(142, 'turnOff');
   end
```

fibaro:getValue(deviceID, propertyName)

Name

Function name must be always the same: **fibaro:getValue**

Application

Gets the current value of the device (deviceID) property (propertyName)

Parameters

deviceID: device ID number

propertyName: name of property

Returned values

String containing current property value

Please note that the return value is of type *string*. When comparing it with a variable of type *number*, use *tonumber* to convert it first.

Code example

```
-- get value of 'brigthness' property (device id = 11)
   local value = fibaro:getValue(11, 'brightness');
```

fibaro:getModificationTime(deviceID, propertyName)

Name

Function name must be always the same: **fibaro:getModificationTime**

Application

Retrieves the status of a property of a device. Specifically, the ‘time last modified’.

Parameters

deviceID: device ID number

propertyName: name of property

Returned values

timestamp: last modification timestamp

Code example

```
-- get time of last 'value' property modification (device id=11)
  local lastModified = fibaro:getModificationTime(11, 'value');
  -- at least 10 seconds after the last modification
  if (os.time() - lastModified >= 10) then
    fibaro:debug('Passed more than 10 seconds');
  else
    fibaro:debug('Passed less than 10 seconds');
  end
```

fibaro:getType(deviceID)

Name

Function name must be always the same: **fibaro:getType**

Application

Gets the type of the device (deviceID)

Parameters

deviceID: device ID number

Returned values

String containing device type (list of types available here)

Code example

```
-- get type of device id = 100
  local type = fibaro:getType(100);
  -- if it's roller shutter
  if (type == 'blind') then
    fibaro:debug('This device is a roller shutter');
  else
    fibaro:debug('Device type: ' .. type);
  end
```

fibaro:getRoomID(deviceID)

Name

Function name must be always the same: **fibaro:getRoomID**

Application

Gets the roomID of a room where selected device (deviceID) is located

Parameters

deviceID: device ID number

Returned values

roomID: number of room where device is located (‘unassigned’ room has roomID equal to zero)

Code example

```
-- Save the room number to 'room' variable (device id = 15)
  local room = fibaro:getRoomID(15);
  if (room == 0) then
    fibaro:debug('This device is \'unassigned\'');
  else
    fibaro:debug('This device is located in: ' .. room);
  end
```

fibaro:getSectionID(deviceID)

Name

Function name must be always the same: **fibaro:getSectionID**

Application

Gets the sectionID of a section where selected device (deviceID) is located

Parameters

deviceID: device ID number

Returned values

sectionID: number of section where device is located (‘unassigned’ section has sectionID equal to zero)

Code example

```
-- Save the section number to 'section' variable (device id = 10)
  local section = fibaro:getSectionID(10);
  if (section == 0) then
    fibaro:debug('This device is unassigned');
  else
    fibaro:debug('This device is located in: ' .. section);
  end
```

fibaro:getDevicesId(filters)

Name

Function name must be always the same: **fibaro:getDevicesId**

Application

Filters all devices in system by given rule.

```
{
  "filter": "hasProperty",
  "value": ["configured", "dead", "model"]
}
{
  "filter": "interface",
  "value": ["zwave", "levelChange"]
}
{
  "filter": "parentId",
  "value": [664]
}
{
  "filter": "type",
  "value": ["com.fibaro.multilevelSwitch"]
}
{
  "filter": "roomID",
  "value": [2, 3]
}
{
  "filter": "baseType",
  "value": ["com.fibaro.binarySwitch"]
}
{
  "filter": "isTypeOf",
  "value": ["com.fibaro.binarySwitch"]
}
{
  "filter": "isPlugin",
  "value": [true]
}
{
  "filter": "propertyEquals",
  "value":
  [
    {
      "propertyName": "configured",
      "propertyValue": [true]
    },
    {
      "propertyName": "dead",
      "propertyValue": [false]
    },
    {
      "propertyName": "deviceIcon",
      "propertyValue": [15]
    },
    {
      "propertyName": "deviceControlType",
      "propertyValue": [15,20,25]
    }
  ]
}
{
  "filter": "deviceID",
  "value": [55,120,902]
}
```

Parameters

filters: filters object

Returned values

devices: array of device id's filtered by given rule.

Code example

```
--[[
%% properties
%% events
%% globals
--]]
local data =
{
  args = { 1 },
  filters =
  {
    {
      filter = "roomID",
      value = { 2 }
    },
    {
      filter = "type",
      value = { "com.fibaro.motionSensor" }
    }
  }
}
local devices = fibaro:getDevicesId(data)
for k,v in ipairs(devices) do
  print (v)
end
```

fibaro:callGroupAction(action, filters)

Name

Function name must be always the same: **fibaro:callGroupAction**

Application

Calls action on devices filtered by given rule.

```
{
  "filter": "hasProperty",
  "value": ["configured", "dead", "model"]
}
{
  "filter": "interface",
  "value": ["zwave", "levelChange"]
}
{
  "filter": "parentId",
  "value": [664]
}
{
  "filter": "type",
  "value": ["com.fibaro.multilevelSwitch"]
}
{
  "filter": "roomID",
  "value": [2, 3]
}
{
  "filter": "baseType",
  "value": ["com.fibaro.binarySwitch"]
}
{
  "filter": "isTypeOf",
  "value": ["com.fibaro.binarySwitch"]
}
{
  "filter": "isPlugin",
  "value": [true]
}
{
  "filter": "propertyEquals",
  "value":
  [
    {
      "propertyName": "configured",
      "propertyValue": [true]
    },
    {
      "propertyName": "dead",
      "propertyValue": [false]
    },
    {
      "propertyName": "deviceIcon",
      "propertyValue": [15]
    },
    {
      "propertyName": "deviceControlType",
      "propertyValue": [15,20,25]
    }
  ]
}
{
  "filter": "deviceID",
  "value": [55,120,902]
}
```

Parameters

action: action name

filters: filters object

Returned values

devices: array of device id's filtered by given rule.

Code example

```
--[[
%% properties
%% events
%% globals
--]]
local data =
{
  args = { 1 },
  filters =
  {
    {
      filter = "roomID",
      value = { 2 }
    },
    {
      filter = "type",
      value = { "com.fibaro.motionSensor" }
    }
  }
}
local devices = fibaro:callGroupAction("setArmed", data)
for k,v in ipairs(devices) do
  print (v)
end
```

Scenes control

fibaro:abort()

Name

Function name must be always the same: **fibaro:abort**

Application

Function stops currently running script.

Parameters

None

Returned values

None

Code example

```
local a = 0;
  while true do
   if (a > 10) then
    fibaro:abort();
   end
   a = a + 1;
   fibaro:sleep(1);
  end
```

fibaro:getSourceTrigger()

Name

Function name must be always the same: **fibaro:getSourceTrigger**

Application

Gets information about the trigger that caused the current scene to run. This function can be also used to determine which of the triggers was the direct cause of the script execution.

Parameters

None

Returned values

An array containing information about the trigger of the current scene.

Returned array includes obligatory 'type' field which, depending on the trigger's type may take the following values:

'property' – for triggers based on the change in device property
'global' – for triggers based on the change in global variable
'other' – other cases (direct run of the scene using Fibaro interface or by another script)

Depending on the 'type' value, the array may have additional fields:

property	global	other
• deviceID triggered device ID number • propertyName name of triggered property	• name name of triggered global variable	• no additional fields

Code example

```
--[[
  %% properties
  13 value
  15 value
  %% globals
  isItDarkOutside
  –]]
  local trigger = fibaro:getSourceTrigger();
  if (trigger['type'] == 'property') then
    fibaro:debug('Source device = ' .. trigger['deviceID']);
  elseif (trigger['type'] == 'global') then
    fibaro:debug('Source global variable = ' .. trigger['name']);
  elseif (trigger['type'] == 'other') then
    fibaro:debug('Other source');
  end
```

fibaro:getSourceTriggerType()

Name

Function name must be always the same: **fibaro:getSourceTriggerType**

Application

Gets information about the type of trigger that caused the current scene to run.

Parameters

None

Returned values

String containing trigger type ('type')

Code example

```
-- Check if scene was triggered manually
  if (fibaro:getSourceTriggerType() == 'other') then
    fibaro:debug('Scene triggered manually');
  end
  -- the same result may be achieved by getting the whole table using getSourceTrigger(
  local source = fibaro:getSourceTrigger();
  if (source['type'] == 'other') then
    fibaro:debug('Scena triggered manually');
  end
```

fibaro:startScene(sceneID)

Name

Function name must be always the same: **fibaro:startScene**

Application

Starts an instance of a given scene (sceneID)

Parameters

sceneID: scene ID number

Returned values

None

Code example

```
  -- if 'a' is greater than 20, run scene (id=10)
     if (a > 20) then
       fibaro:startScene(10);
     end
```

fibaro:killScenes(sceneID)

Name

Function name must be always the same: **fibaro:killScenes**

Application

Terminates all running instances of a given scene (sceneID)

Parameters

sceneID: scene ID number

Returned values

None

Code example

```
  -- If the value of 'a' variable is in the range [1;5], terminate all instances of scene (id=2)
     if (a &gt;= 1 and a &lt;= 5) then
       fibaro:killScenes(2);
     end
```

fibaro:setSceneEnabled(sceneID, enabled)

Name

Function name must be always the same: **fibaro:setSceneEnabled**

Application

Activates/deactivates a scene with a given sceneID

Parameters

sceneID: scene ID number

enabled: boolean value (true – enable scene, false – disable scene)

Returned values

None

Code example

```
-- If variable 'a' is positive, deactivate scene (id=3)
   if (a>0) then
     fibaro:setSceneEnabled(3, false);
   -- otherwise activate scene (id=3)
   else
     fibaro:setSceneEnabled(3, true);
   end
```

fibaro:isSceneEnabled(sceneID)

Name

Function name must be always the same: **fibaro:isSceneEnabled**

Application

Checks if the scene with a given sceneID is active.

Parameters

sceneID: scene ID number

Returned values

Boolean value**:** true – scene enabled, false – scene disabled

Code example

```
-- if one scene (id=3) is active, enable also another scene (id=5)
   if (fibaro:isSceneEnabled(3)) then
     fibaro:setSceneEnabled(5, true);
   end
```

fibaro:countScenes()

Name

Function name must be always the same: **fibaro:countScenes**

Application

Gets the number of currently running instances of a scene.

Parameters

None

Returned values

A positive integer containing the number of currently active scenes.

Code example

```
-- number of running scenes
  local num = fibaro:countScenes();
  if (num == 1) then
    fibaro:debug('one running scene');
  else
    fibaro:debug('Number of running scenes: ' .. num);
  end
  -- max one instance of specified scene
  if (fibaro:countScenes() > 1) then
    fibaro:abort();
  end
```

fibaro:countScenes(sceneID)

Name

Function name must be always the same: **fibaro:countScenes**

Application

Gets the number of currently running instances of a given scene.

Parameters

sceneID: scene ID number

Returned values

A positive integer that specifies the number of currently active scenes.

Code example

```
-- number of running instances of scene id=12
  local num = fibaro:countScenes(12);
  fibaro:debug('Number of running instances of scene id=12 is : ' .. num);
  -- Check if scene id=12 is running
  if (fibaro:countScenes(12) >= 1) then
    fibaro:debug('Scene is running');
  else
    fibaro:debug('Scene is not running');
  end
```

fibaro:setSceneRunConfig(sceneID,runConfig)

Name

Function name must be always the same: **fibaro:setSceneRunConfig**

Application

Where runConfig is string that takes one of these three values:

- TRIGGER_AND_MANUAL
- MANUAL_ONLY
- DISABLED

Which will set given scene respectively to Automatic, Manual and Disabled mode. Any other value will set scene triggering mode to Automatic.

Parameters

sceneID: scene ID number

runConfig: string value (TRIGGER_AND_MANUAL, MANUAL_ONLY, DISABLED)

Returned values

None

Code example

```
-- set run config element
  fibaro:setSceneRunConfig(123, 'TRIGGER_AND_MANUAL');
```

fibaro:getSceneRunConfig(sceneID)

Name

Function name must be always the same: **fibaro:getSceneRunConfig**

Application

Returns currently set value.

Parameters

sceneID: scene ID number

Returned values

String containing runConfig value

Code example

```
-- get run config element
  fibaro:getSceneRunConfig(123);
```

Global variables management

fibaro:getGlobal(varName)

Name

Function name must be always the same: **fibaro:getGlobal**

Application

Retrieves the ‘value’ and ‘time last modified’ of a global variable registered in the Variables Panel.

Parameters

varName: name of global variable

Returned values

string containing current global variable value

timestamp: last modification timestamp

Please note that these return values are of type *string*. When comparing them with a variable of type *number*, use *tonumber* to convert them first.

Code example

```
-- get a value and time of the last "isNight" global variable modification
  local value, modificationTime = fibaro:getGlobal('isNight');
  -- second value may be omitted
  local value2 = fibaro:getGlobal('isNight');
  -- returned value may be used as a scene condition
  if (value == '1') then
    fibaro:debug('It's night!');
  end
```

fibaro:getGlobalModificationTime(varName)

Name

Function name must be always the same: **fibaro:getGlobalModificationTime**

Application

Retrieves the 'time last modified' of a global variable registered in the Variables Panel.

Parameters

varName: name of global variable

Returned values

timestamp: last modification timestamp

Please note that this return value is of type *string*. When comparing it with a variable of type *number*, use *tonumber* to convert it first.

Code example

```
-- get time of last 'counter' variable modification
  local lastModified = fibaro:getGlobalModificationTime('counter');
  -- at least 10 seconds after the last modification
  if (os.time() - lastModified >= 10) then
    fibaro:debug('Passed more than 10 seconds');
  else
    fibaro:debug('Passed less than 10 seconds');
  end
```

fibaro:getGlobalValue(varName)

Name

Function name must be always the same: **fibaro:getGlobalValue**

Application

Retrieves the 'value' of a global variable registered in the Variables Panel.

Parameters

varName: name of global variable

Returned values

string containing current global variable value

Please note that this return value is of type *string*. When comparing it with a variable of type *number*, use *tonumber* to convert it first.

Code example

```
-- get value of 'counter' global variable
   local counterValue = fibaro:getGlobalValue('counter');
```

fibaro:setGlobal(varName, value)

Name

Function name must be always the same: **fibaro:setGlobal**

Application

Changes the value of a global variable

Parameters

varName: name of global variable

value: new value of global variable

Returned values

None

Code example

```
-- Assign value '1' to 'index' global variable
   fibaro:setGlobal('index', 1);
   -- Increase value of 'test' global variable by adding 3
   fibaro:setGlobal('test', fibaro:getGlobalValue('test') + 3);
   -- Assign value of 'a' local variable to 'index' global variable
   local a = 10 * 234;
   fibaro:setGlobal('index', a);
```

Additional functions

fibaro:calculateDistance(position1, position2)

Name

Function name must be always the same: **fibaro:calculateDistance**

Application

Function calculates distance between two geographical points: position1 and position2.

Parameters

position1: the first location

position2: the second location

Points are defined using geographical coordinates. Their values are expressed in degrees with a decimal part and separated by a semicolon. Negative values are, respectively, west and south. The decimal separator is a dot.

For example: 40°44′55″N, 73°59′11″W = „40.7486;-73.9864”

Returned values

Distance in meters

Code example

```
local userLocation = fibaro:getValue(123, 'Location');
   local exampleLocation = "52.4325295140701;16.8450629997253";
   local result;
   result = fibaro:calculateDistance(userLocation, exampleLocation);
   fibaro:debug('Distance is ' .. result .. 'm.');
```

fibaro:debug(text)

Name

Function name must be always the same: **fibaro:debug**

Application

Outputs a string to the debug console associated with the script.

Parameters

text: text to be displayed

Returned values

None

Code example

```
fibaro:debug('Example text');
```

fibaro:sleep(time)

Name

Function name must be always the same: **fibaro:sleep**

Application

Suspends execution of a script for a specified time in milliseconds.

Parameters

time: number of miliseconds

Returned values

None

Code example

```
-- Wait 10 seconds
   fibaro:sleep(10000);
```

HomeCenter.SystemService.reboot()

Name

Function name must be always the same: **HomeCenter.SystemService.reboot()**

Requirements

- Fibaro Home Center 2 updated to 4.081 software version or higher

Application

Reboots system.

Parameters

None

Returned values

None

Code example

```
-- Reboot system
   HomeCenter.SystemService.reboot();
```

HomeCenter.SystemService.shutdown()

Name

Function name must be always the same: **HomeCenter.SystemService.shutdown()**

Requirements

- Fibaro Home Center 2 updated to 4.081 software version or higher

Application

Shutdowns system.

Parameters

None

Returned values

None

Code example

```
-- Shutdown system
   HomeCenter.SystemService.shutdown();
```

Plugins control

function onAction(deviceId, action)

Name

Function name must be always the same: **onAction**

Application

Function is usually used for handling an event assigned to the elements (button, select, etc.) at "Advanced" tab. However this function may be called via any http client by constructing a request as described here. Users of our system may also call such function for example by scenes. Let us assume that we have lightControl plugin, which is able to switch off a light. Plugin contains a method: switchOff(LightID). Scene developer may call function switchOff(LightID) by sending the POST request: POST api/devices/DEVICE_ID/action/ACTION_NAME {"args":["arg1", ..., "argN"]}

Location

Function has to be implemented in file: **main.lua**

Trigger

Function is triggered when RESTful callAction request (described here) is called. For plugin developer, it's important that it may be triggered as a result of pushing the button at "Advanced Settings" plugin's tab. Function onAction may be also called from one of the scenes or using any client which is able to construct http requests to our server (main controller).

Parameters

1. **deviceId** - plugin's ID (device or plugin) distinctive for this resource
2. **action** - array containing:
    1. **actionName** - (action.actionName) string containing name of the function you want to run (e.g. by pushing the button at "Advanced" tab). Generally, function is set in PluginName.lua file.
    2. **args** - (action.args) it's an array containing the list of additional arguments

Code example

```
function onAction(deviceId, action)
      return alphatechFarfisa:callAction(action.actionName, unpack(action.args))
   end
```

function onUIEvent(deviceId, event)

Name

Function name must be always the same: **onUIEvent**

Application

Function is usually used for handling an event assigned to one of the handlers (button, select, switch, etc.) at "General" tab.

Location

Function has to be implemented in file: **main.lua**

Trigger

Function is triggered as a result of plugin's handler usage (button or slider) from "General" tab (called "View").

Parameters

1. **deviceId** - plugin's ID (device or plugin) distinctive for this resource
2. **event** - array containing:
    1. **elementName** - (event.elementName) string containing handler's name (name="handlerName" – defined in view.xml). This allows to know which GUI component (from "General" tab) has just been used. Furthermore, it enables running appropriate function (bounded with that name) implemented usually in PluginName.lua file.

       ```
       uiBinding = {
               ["POWER_Button"] = function() lightControl:sendKey("KEY_POWEROFF
               ["NUM_1_Button"] = function() lightControl:sendKey("KEY_1") end,
               ["NUM_2_Button"] = function() lightControl:sendKey("KEY_2") end
           }

           function onUIEvent(deviceId, event)
               local callback = uiBinding[event.elementName]

               if (callback) then
                   callback(event)
               end
           end
       ```

    2. **deviceId –** number - plugin's ID (device or plugin) distinctive for this resource

    3. **eventType** – string containing event's name, e.g. onToggled, onPressed, onChanged, etc. This allows to know which handler's type was used and what action should be taken.
       Example: if we use the class supporting holding down the button, we get the info when this button is pressed (onPressed) and when it's released (onReleased).

       ```
       if (event.eventType == 'onReleased') then
                   self:mouseUp(event)
               elseif (event.eventType == 'onPressed') then
                   self:mouseDown(event)
               elseif (event.eventType == 'cancel') then
                   self:cancelEvents()
               end
           end
       ```

    4. **values** - array containing e.g. slider's value, bool value (false/true) for relay switch, etc.

Code example

```
function onUIEvent(deviceId, event)
        local callback = uiBinding[event.elementName]

        if (callback) then
            callback(event)
        end
    end
```

function configure(deviceId, config)

Name

Function name must be always the same: **configure**

Application

This function is used for handling "Save" event performed by the user at plugin's "General" tab. Generally, it's applied to make sure that all plugin properties are in line with our requirements. Otherwise we can react for such a situation by changing the property value.

Location

Function has to be implemented in file: **main.lua**

Trigger

Function is triggered when "Save" button from plugin's "General" tab (called "View") is clicked.

Parameters

1. **deviceId** - plugin's ID (device or plugin) distinctive for this resource
2. **config** - array containing a list of all current plugin's properties.

Code example

```
function configure(deviceId, config)
      if deviceId == lightControl.id then
         restart = false
         if (config.ip) then
            lightControl:updateProperty('ip', config.ip)
            restart = true
         end
         if (config.pollingTimeout) then
            lightControl:updateProperty('pollingTimeout', config.pollingTimeout)
         end
         if (restart) then
            plugin.restart()
         end
      end
   end
```

function updateProperty(name, value)

Name

Function name must be always the same: **updateProperty**

Application

This function is used for saving plugin's property.
Example: If plugin has a property called "userName" = "Tom", using this function we can change its value for any other string value.

Location

Function is implemented in device.lua. On the other hand, function call may be performed in various files, e.g.: **main.lua, PluginName.lua**

Trigger

Function is triggered at the request of the developer (anywhere in the code).

Parameters

1. **name** - string containing the name of updated property
2. **value** - (variable supporting various types: string, int, float, etc.) new value assigned for the property given in the "name" parameter.

Code example

```
lightControl:updateProperty('userName', 'Marcin')
```

function setName(name)

Name

Function name must be always the same: **setName**

Application

This is function is used for assigning new value for the "name" plugin's property.

Location

Function is implemented in device.lua. On the other hand, function call may be performed in various files, e.g.: **main.lua, PluginName.lua**

Trigger

Function is triggered at the request of the developer (anywhere in the code).

Parameters

1. **name** - string containing value for the "name" property we want to update.

Code example

```
function Device:setName("multiSensor")
```

function setEnabled(enabled)

Name

Function name must be always the same: **setEnabled**

Application

This function is used for saving "enabled" plugin's property. If it's set to false, selected device/plugin is visible in the system, but can't be controlled. Setting to true enables its control.

Location

Function is implemented in device.lua. On the other hand, function call may be performed in various files, e.g.: **main.lua, PluginName.lua**

Trigger

Function is triggered at the request of the developer (anywhere in the code).

Parameters

1. **enabled** - boolean state (true/false) of "enabled" property we want to update.

Code example

```
function Device:setEnabled(true)
```

function setVisible(visible)

Name

Function name must be always the same: **setVisible**

Application

This function is used for saving “visible” plugin’s property. If it’s set to false, selected device/plugin is invisible in the system. Setting to true enables its visibility.

Location

Function is implemented in device.lua. On the other hand, function call may be performed in various files, e.g.: **main.lua, PluginName.lua**

Trigger

Function is triggered at the request of the developer (anywhere in the code).

Parameters

1. **visible** - boolean state (true/false) of “visible” property we want to update.

Code example

```
function Device:setVisible(visible)
```

function getDevice(deviceId)

Name

Function name must be always the same: **getDevice**

Application

This function returns all available properties for selected device (ID given in deviceId parameter).

Location

Function is implemented in plugin.lua. On the other hand, function call may be performed in various files, e.g.: **main.lua, PluginName.lua**

Trigger

Function is triggered at the request of the developer (anywhere in the code).

Parameters

1. **deviceId** - number - plugin’s ID (device or plugin) distinctive for this resource.

Code example

```
function plugin.getDevice(15)
```

function getProperty(deviceId, propertyName)

Name

Function name must be always the same: **getProperty**

Application

This function returns value of the specified property given in propertyName parameter.

Location

Function is implemented in plugin.lua. On the other hand, function call may be performed in various files, e.g.: **main.lua, PluginName.lua**

Trigger

Function is triggered at the request of the developer (anywhere in the code).

Parameters

1. **deviceId** - number - plugin's ID (device or plugin) distinctive for this resource.
2. **propertyName** - string containing name of the property which value we want to get.

Code example

```
function plugin.getProperty(15, 'name')
```

function getChildDevices(deviceId)

Name

Function name must be always the same: **getChildDevices**

Application

This function returns an array containing all child devices (owning "parentId" property which is equal to parent's "deviceId" property) of selected plugin.

Location

Function is implemented in plugin.lua. On the other hand, function call may be performed in various files, e.g.: **main.lua, PluginName.lua**

Trigger

Function is triggered at the request of the developer (anywhere in the code).

Parameters

1. **deviceId** - number - plugin's ID (device or plugin) distinctive for this resource.

Code example

```
function plugin.getChildDevices(15)
```

function createChildDevice(parentId, type, name)

Name

Function name must be always the same: **createChildDevice**

Application

This function creates new child device for selected plugin and returns all available properties as an array

Location

Function is implemented in plugin.lua. On the other hand, function call may be performed in various files, e.g.: **main.lua, PluginName.lua**

Trigger

Function is triggered at the request of the developer (anywhere in the code).

Parameters

1. **parentId** - number – plugin's ID (device or plugin) for which you want to create a child device
2. **type** – string – device type, e.g. com.fibaro.xyz for created child device
3. **name** – string – name of created child device

Code example

```
function plugin.createChildDevice(15, 'com.fibaro.lightControlLight', 'lamp1')
```

function restart()

Name

Function name must be always the same: **restart**

Application

This function restarts plugin's LuaEnvironment process. It's used to recall plugin's initialization process.

Location

Function is implemented in plugin.lua. On the other hand, function call may be performed in various files, e.g.: **main.lua, PluginName.lua**

Trigger

Function is triggered at the request of the developer (anywhere in the code).

Code example

```
function plugin.restart()
```

TCPListener Class Reference

Inheritance

Inherits from **Object:**

```
class 'TCPListener' (Object)
```

Dependencies

```
require('common.Object')
   require('net.TCPSocket')
```

Overview

Handles asynchronous read of data from TCP socket. Provides interaction with events:

- **error:** called when error occures
- **connected:** when socket is succesfully connected
- **disconnected:** when socked is disconnected
- **dataReceived:** when same data were succesfully read from socket

Events can be registered with registerEvent(eventName, handler) method.
Upon succesfull connection it starts to listen on socket

Properties

delimiter

- **type:** string
- **discution:** Data received will be sliced by delimiter. If not provided, dataReceived event would be rised after every succesfull read.

```
self.delimiter = options.delimiter or ''
```

connectTimeout

- **type:** integer
- **discution:** Number of millisecond after which connection will be dropped in case of lack in response.

```
self.connectTimeout = options.connectTimeout or 3000
```

readTimeout

- **type:** integer
- **discution:** Number of millisecond after which waiting for data to received will be dropped in case of lack in response.

```
self.readTimeout = options.readTimeout or 3000
```

sock

- **type:** net.TCPSocket object
- **discution:** Responsible for sending and receiving data through socket..

```
self.sock = net.TCPSocket()
```

Methods

__init(options)

- **overview:** Contstructor.
- **params:**
  - **options:**
    - **type:** array
    - **discution:** Optional parameter. If not empty, should be an array(key => value) with pairs (delimiter => value, connectionTimeou t=> value, readTimeout => vale).

close()

- **overview:** Closes tcp connection.

send(data)

- **overview:** Sends data to socket.
- **params:**
  - **data:**
    - **type:** string
    - **discution:** Data to be sent.

__succes()

- **overview:** Called after a successful reading data from the socket.
- **params:**
  - **data:**
    - **type:** string
    - **discution:** Data received from socket.

__error(err)

- **overview:** Called after a failed reading data from the socket.
- **params:**
  - **err:**
    - **type:** string
    - **discution:** Data received from socket.

Use example

```
self.sock = TCPListener({delimiter = string.char( 0x0D , 0x0A ), readTimeout = 10000, co
    self.sock:registerEvent('dataReceived', function(sock, data) self:onDataReceived(data,
    self.sock:registerEvent('disconnected', function()  self:onClosed() end)
    self.sock:registerEvent('connected', function() self:onConnected() end)
    self.sock:registerEvent('error', function(sock, error) print("error", error) end)
```

HTTPClient Class Reference

Overview

The HTTPClient class provides client side of the HTTP protocol.

To use this class you should require it in pluginName.lua

```
require('net.HTTPClient')
```

Methods

HTTPClient(timeout)

- **overview:** Contstructor.
- **params:**
  - **timeout:**
    - **type:** array
    - **discution:** Number of millisecond after which connection will be dropped in case of lack in response.

```
self.http = net.HTTPClient({timeout=2000})
```

request(url, options, succes, error)

- **overview:** Sends http request to server.
- **params:**
  - **url:**
    - **type:** string
    - **discution:** Url of the the server.
  - **options:**
    - **type:** array
    - **discution:** Array(key => value) with pairs (headers => value, data => value, method => vale, timeout => value).
  - **succes:**
    - **type:** callback function
    - **discution:**Called after a successful connection.
  - **error:**
    - **type:** callback function
    - **discution:**Called after a faild connection.

```
self.http:request(controlUrl, {
        options={
            headers = self.controlHeaders,
            data = requestBody,
            method = 'POST',
            timeout = 5000
        },
        success = function(status)
            local result = json.decode(status.data)
            if result[1] then
                if result[1].error then
                    print ("ERROR")
                    print ("  type: " .. result[1].error.type)
                    print ("  address: " .. result[1].error.address)
                    print ("  description: " .. result[1].error.description)
                elseif result[1].success then
                    self:updateProperty("userName", result[1].success.username)
                else
                    print ("unknown response structure: ")
                    print(status)
                end
            else
                print ("unknown response structure: ")
                print(status)
            end
        end,
        error = function(error)
            print "ERROR"
            print(error)
        end
    })
```

UDPSocket Class Reference

Overview

The FUdpSocket class provides a UDP socket.

To use this class you should require it in pluginName.lua

```
require('net.UDPSocket')
```

Methods

UDPSocket(broadcast)

- **overview:** Contstructor.
- **params:**
  - **broadcast:**
    - **type:** boolean
    - **discution:** Define if broadcast mode should be used.

```
self.udp = net.UDPSocket({ broadcast = true})
```

sendTo(payload, ip, port)

- **overview:** Sends data to given IP and port.
- **params:**
  - **payload:**
    - **type:** string
    - **discution:** Data to send.
  - **ip:**
    - **type:** string
    - **discution:** Destination address.
  - **port:**
    - **type:** integer
    - **discution:** Destination port.

```
self.udp:sendTo(payload, '255.255.255.255', 9,{
        success = function()
           print('Secces:')
        end,
        error = function(error)
           print('Error:', error)
        end
     })
```

Popup service

HomeCenter.PopupService.publish({title, subtitle, contentTitle, contentBody, img, type, buttons})

Name

Function name must be always the same: **HomeCenter.PopupService.publish**

Application

This function is used for creating pop-ups to be displayed on mobile devices. This way you can get a customizable notification of any event and/or trigger a scene using the button located in the pop-up window.

Requirements

- Fibaro Home Center 2 updated to 4.045 software version or higher
- Fibaro mobile application:
  - Fibaro for iPhone 2.5 or higher
  - Fibaro for iPad 1.50 or higher
  - Fibaro for Android phones 1.6.0 or higher
  - Fibaro for Tablet 1.3.0 or higher

Parameters

1. **title** - string containing text to be displayed as a pop-up window title (parameter required)
2. **subtitle** - string containing text to be displayed as a pop-up window subtitle
3. **contentTitle** - string containing text to be displayed as a pop-up content title
4. **contentBody** - string containing text to be displayed as a pop-up content (parameter required)
5. **img** - string containing path of an image to be displayed in the pop-up window (supported extensions: .jpg, .bmp, .png, .gif)
6. **type** - notification type indicated with a colour, available types:
   - 'Info' - blue (default)
   - 'Success' - green
   - 'Warning' - yellow
   - 'Critical' - red
7. **buttons** - array containing definitions of buttons to be displayed in the pop-up, single button definition must be an array containing:
   - 'caption' - text displayed on the button
   - 'sceneId' - scene id triggered after pushing the button

At most 3 buttons may be defined. There is no need to create any button - ‘ok’ button will be created by default.

1st Code example

```
--[[
%% properties
%% globals
--]]
-- variable containing path of Motion Sensor’s icon
local imgUrl =
'http://www.fibaro.com/sites/all/themes/fibaro/images/motion-
sensor/en/motion_sensor_manual.png'
-- pop-up call
HomeCenter.PopupService.publish({
      -- title (required)
   title = 'No motion detected',
      -- subtitle(optional), e.g. time and date of the pop-up call
   subtitle = os.date("%I:%M:%S %p | %B %d, %Y"),
      -- content header (optional)
   contentTitle = 'No motion since last 15 minutes',
      -- content (required)
   contentBody = 'Should I run the scene "Night"?',
      -- notification image (assigned from the variable)
   img = imgUrl,
      -- type of the pop-up
   type = 'Success',
      -- buttons definition
   buttons = {
   { caption = 'Yes', sceneId = 0 },
   { caption = 'No', sceneId = 0 }
   }
})
```

NOTE
Please note that the example scene must be **triggered manually.** It just illustrates the way of creating pop-ups. Execution of this scene will not affect any device status (sceneId = 0).

NOTE
Setting an action of the button to **'sceneId = 0'** means that no action will be performed.

NOTE
Created pop-up is sent to **each of users and mobile devices** connected with the main controller.

NOTE
There is no maximum size of image displayed in the pop-up window. However, using **too large** file may result in long waiting times required for downloading the image.

NOTE
Pushing one of the buttons displayed in the pop-up window may only trigger **another scene.**

2nd Code example

Scene generating pop-up after meeting a specified condition

```
--[[
%% properties
3814 value
%% globals
--]]
local startSource = fibaro:getSourceTrigger();
if (
    ( tonumber(fibaro:getValue(3814, "value")) > 60 )
or
startSource["type"] == "other"
)
then
HomeCenter.PopupService.publish({
    title = 'Brightness level',
    subtitle = 'is too high',
    contentTitle = 'Dimmer',
    contentBody = 'Would you like to turn it off?',
    img = ' http://www.fibaro.com/images/eng/icon_osw.png',
    type = 'Critical',
    buttons = {
        { caption = 'Turn off', sceneId = 3228 },
        { caption = 'No', sceneId = 0 },
        { caption = 'Set to 100%', sceneId = 3229 }
    }
})
end
```

Notifications control

HomeCenter.NotificationService.publish(payload)

Name

Function name must be always the same: **HomeCenter.NotificationService.publish**

Application

Publishes notification.

Parameters

request: request object

Code example

```
HomeCenter.NotificationService.publish({
    type = "GenericDeviceNotification",
    priority = "warning",
    data =
    {
        deviceId = 2643,
        title = "foo",
        text =  "bar"
    }
})
```

HomeCenter.NotificationService.update(id, payload)

Name

Function name must be always the same: **HomeCenter.NotificationService.update**

Application

Updates notification.

Parameters

id: notification id

request: request object

Code example

```
HomeCenter.NotificationService.update(7, {
    canBeDeleted = true,
    data =
    {
        title = "udapted foo",
        text =  "udapted bar"
    }
})
```

HomeCenter.NotificationService.remove(id)

Name

Function name must be always the same: **HomeCenter.NotificationService.remove**

Application

Removes notification.

Parameters

id: notification id

Code example

```
HomeCenter.NotificationService.remove(7)
```