

2.4

ekinex
CONTROL YOUR LIVING SPACE

LUA Scripting Programming Manual



DELÉGO SERVER Visualisation webserver

EK-DEL-SRV01 / EK-DEL-SRV01-M

Index

1 Introduction.....	4
1.1 Scope of this manual.....	4
1.2 Conventions.....	4
1.3 Safety information.....	4
1.4 Warnings.....	5
1.5 Copyright.....	5
1.6 Open Source Licenses.....	5
1.7 More information.....	5
2 Overview.....	7
2.1 Scripts.....	7
2.2 Runners.....	7
2.3 Libraries.....	8
2.4 Inputs and Outputs.....	9
2.5 Inclusion of scripts.....	10
2.6 Variable inspection.....	10
2.7 Tables.....	11
2.7.1 Number of elements.....	11
2.7.2 Cycle Between Items.....	11
3 Scripts.....	12
3.1 Introduction.....	12
3.2 List.....	12
3.3 Editor.....	13
3.4 Examples.....	14
4 Runners.....	15
4.1 Introduction.....	15
4.2 Adding a new runner.....	15
4.2.1 Inputs.....	16
4.2.2 Outputs.....	17
4.3 Loop.....	17
4.3.1 Cyclic Execution.....	17
4.3.2 Execution At Interrupt.....	18
4.4 Debugging.....	19
5 Models.....	20
5.1 Introduction.....	20
5.2 Creating a model.....	20
6 Objects Library.....	24
6.1 Introduction.....	24
6.2 Inclusion.....	24
6.3 Loading Objects.....	24
6.4 Object Properties.....	25
6.5 Commands.....	27
6.6 Search For Objects.....	28
6.7 Search by address.....	28
6.8 Editing objects.....	29
7 Surrounding Library.....	30

7.1 Introduction.....	30
7.2 Inclusion.....	30
7.3 Inputs and Outputs.....	30
7.4 Identifier.....	31
8 Messages library.....	32
8.1 Introduction.....	32
8.2 Inclusion.....	32
8.3 Video Notifications.....	32
8.4 Mail Notifications.....	32
9 JSON library.....	34
9.1 Introduction.....	34
9.2 Inclusion.....	34
9.3 Decoding.....	34
9.4 Encoding.....	34
9.5 Credits.....	34
10 URL library.....	35
10.1 Introduction.....	35
10.2 Inclusion.....	35
10.3 Getting URLs.....	35
11 Appendices.....	36
11.1 Functional Types.....	36
11.2 Web Types (Phpclass).....	37
11 Revision history.....	39

1 Introduction

1.1 Scope of this manual

This document contains all the information to create your own scripts in DELÉGO SERVER using the LUA programming language.

The present user manual is part of the suite of application manuals dedicated to the DELÉGO SERVER, available on the section dedicated to this product in the EKINEX website. Before reading this document, it is recommended to consult the following application manuals:

REF	DOCUMENT	DESCRIPTION
[D1]	DOCUMENTATION PLAN	Overview of the application manuals of DELÉGO SERVER
[D2]	GENERAL FUNCTIONS - APPLICATION USER MANUAL	General installation manual of the webserver
[D3]	AUTOMATIONS - SYSTEM CONFIGURATION MANUAL	Connections and logics

These documents are available for download in the area of the EKINEX website dedicated to the webserver.

1.2 Conventions

The following conventions are used in this manual:

[KEY]	The keyboard keys that the installer must press are shown in square brackets
<i>Courier</i>	Software messages generated by the DELÉGO SERVER configuration software are reported in the "courier" font
<i>Function name</i>	Names of buttons, table fields or other graphic elements of the user interface of the DELÉGO SERVER configuration software are shown in italics
	Information note
	Important note that requires particular attention

1.3 Safety information

DELÉGO SERVER is a product intended for trained and specialized personnel only; anyone who interacts with the device must first read this documentation, especially this safety information. This document supplements and does not replace any legislation or legal directives on safety.

The device has been developed following the safety regulations currently in force; however, it is not possible to exclude with certainty possible damage or interactions with other devices during its operation. The device complies with EMC guidelines and harmonized European standards; any modifications to the device may affect EMC compatibility.

The supply voltage must be strictly within the range indicated in the GENERAL FUNCTIONS application manual and on the device; risk of fire or explosion if powered outside this range. The CE declaration of conformity of the device can be requested from EKINEX®, at the references on the site www.ekinex-delego.com.



In compliance with Directive 2002/96/EC the DELÉGO SERVER electronic device must be disposed of in the appropriate facilities and not in the collection of solid urban waste.

1.4 Warnings

- This application manual is aimed at installers, system integrators and designers.
- The assembly, electrical connection, configuration and commissioning of the appliance can only be carried out by specialized personnel in compliance with the applicable technical standards and the laws in force in the respective countries
- Opening the appliance case causes the immediate interruption of the warranty period
- In the event of tampering, compliance with the essential requirements of the applicable directives for which the appliance has been certified is no longer guaranteed.
- EKINEX defective appliances must be returned to the manufacturer at the following address:
Ekinex S.p.A. Via Novara 37, 28010 Vaprio d'Agogna (NO)

1.5 Copyright

© Ekinex S.p.A. 2025

This application manual, the DELÉGO SERVER software and the DELÉGO iOS and Android apps are subject to copyright; all rights are reserved. Copying, reproduction, translation and/or modification, even partial, are expressly prohibited unless approved in writing by EKINEX®.

1.6 Open Source Licenses

DELÉGO SERVER contains open source software, such as the Linux operating system and its kernel. These software components are subject to various open-source licenses, including:

- GNU General Public License (GPL), <https://www.gnu.org/licenses/gpl-2.0.html>
- GNU Lesser General Public License (LGPL), <https://www.gnu.org/licenses/lgpl-2.0.en.html>

If you own a product, for a period of 3 years from the last production date, you can request from EKINEX the source code of the software components licensed under the GNU General Public License (GPL) or the GNU Lesser General Public License (LGPL), and use, distribute and modify them in accordance with the respective licenses.

EKINEX cannot be considered responsible in any way for the source code thus distributed, which is provided without guarantees. EKINEX cannot also be held responsible for any damage or consequence resulting from modifications (additions / removals) made to this software by third parties, unless specifically authorized.

1.7 More information

- For further information on the product, you can contact EKINEX technical support® to the e-mail address: support@EKINEX.com or consult the website www.EKINEX.com

- EKINEX® is a registered trademark of Ekinex S.p.A.
- KNX® ed ETS® are registered trademarks of KNX Association cvba, Brussels

The company reserves the right to make changes to this documentation without notice.

2 Overview

2.1 Scripts

The SCRIPTS are files containing code that can be created and stored on the DELÉGO SERVER flash disk through a special section in the administration space. These files can be created, modified and deleted through an integrated editor.

The files remain saved on the DELÉGO SERVER disk and persist even after a reset of the factory parameters; on the other hand, they are not included in the DELÉGO SERVER backup of the database, therefore they must be exported (and subsequently imported) separately if you want to transfer the project to another webserver.

The SCRIPTS must be created following the instructions contained in this manual, making use of the libraries and functions made available by the scripting engine; any code that goes beyond these indications can potentially compromise the correct functioning of DELÉGO SERVER.

DELÉGO SERVER is already equipped with a series of examples that offer a complete overview of the potential offered by the scripting system; you are not intended to modify or execute these examples directly, but you may duplicate them and work on the copy without any restrictions.

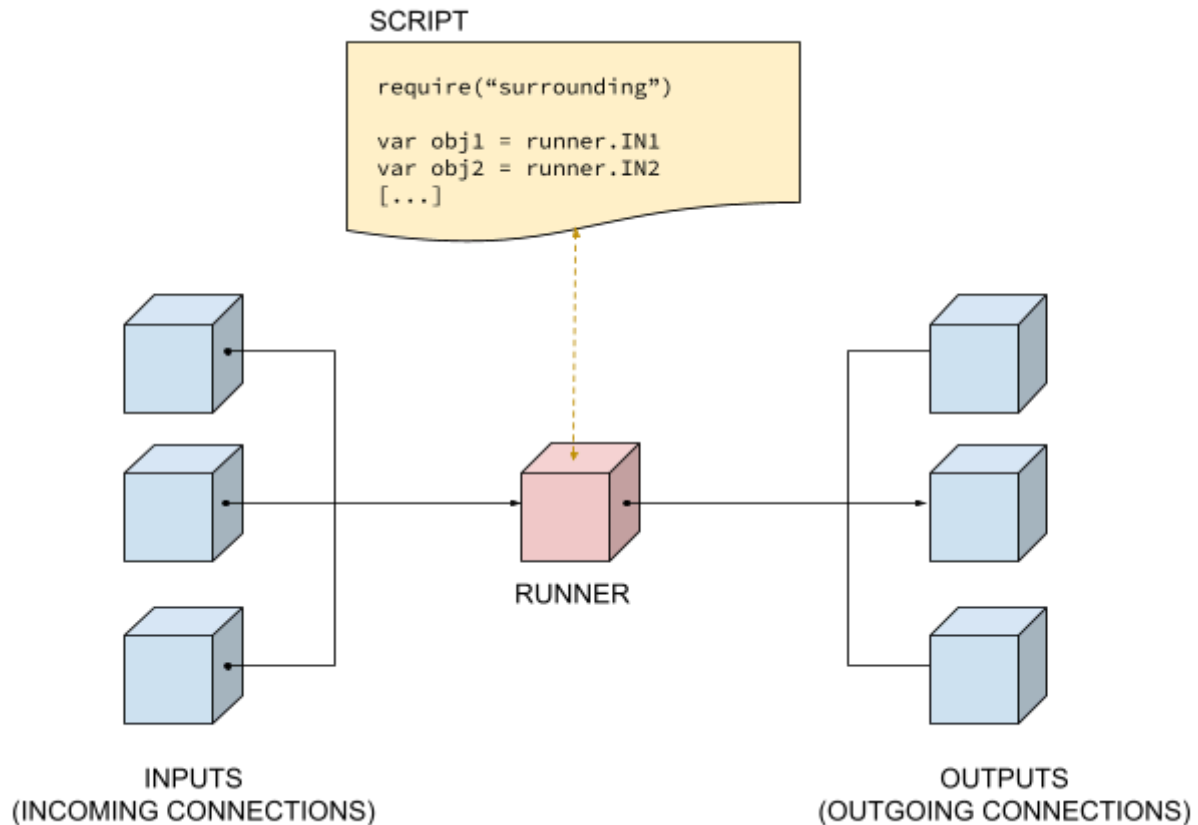
2.2 Runners

Once a script has been created (from scratch or by duplicating and modifying an example), it is necessary to associate it with one or more objects of the "script runner" type; these particular objects have the task of executing the code contained in the script to which they are associated, when connected to other objects in the project.

The SCRIPTS cannot therefore be executed directly, but only through a SCRIPT RUNNER, which behaves in a completely analogous way to other types of object. The advantage of having a structure based on separate SCRIPTS and SCRIPT RUNNERS is twofold:

- Possibility of associating the same SCRIPT to more than one RUNNER object, in turn inserted in different logics, events, scenarios, etc... and therefore able to manipulate different data
- Ability to export, import and share scripts regardless of the project created

The following diagram illustrates in principle the interactions between a SCRIPT RUNNER and the associated objects via EVENTS, or its command via the RUNTIME pages of the webserver:



2.3 Libraries

DELÉGO SERVER provides a series of libraries, which offer ready-made tools for interacting with the main aspects covered by the scripting engine. Libraries are not automatically included in scripts, but must be called using the LUA include command

```
require(<library name>)
```

Example:

```
require("messages")
```

Since the inclusion of each library involves a processing time, it is recommended to include only the libraries that are strictly necessary for the task that the script must carry out. Libraries can in turn depend on other libraries; in this case, it is not necessary to also include the related libraries, as the inclusion is done automatically by calling the library that has the dependencies.

The following table shows a list of the available LUA libraries, a brief description and any dependencies; in the following chapters they will be examined in detail.

Library	Description
object	If offers methods to load and interacts with objects
surrounding	It offers a set of objects to interact with the SCRIPT RUNNER object that is executing the script and the related INPUTS and OUTPUTS
messages	It allows you to send video, email and push notifications
json	If offers methods to encode and decode JSON strings
url	It allows to call an external URL and handle the response

2.4 Inputs and Outputs

When the SCRIPT RUNNER object is executed, it always receives an INPUT value; in particular:

- In the case of direct command from the FRONTEND interface, the value is given by the user input (e.g. by typing a value in a text box, or by pressing a button - it depends on the graphic aspect chosen for the SCRIPT RUNNER)
- In the case of a command following the change of state of one of the "parent" objects associated as INPUT CONNECTIONS to the SCRIPT RUNNER, the input value is given by the properties of the event itself, as defined in administration. In particular, it is possible to pass as a value:
 - The value of the object that changes state, causing the event to be executed
 - A constant value

In both cases, the input value is available within the script by calling the function

```
input()
```

Example:

```
val = input()
```

At the end of the execution, the script can return an OUTPUT value to the execution environment; if the RUNNER is related to one or more objects in output, the value is set to these objects.

To return a value to the execution environment, simply call the function

```
output()
```

Example:

```
output(1)
```



Calling the `output()` function terminates script execution; any subsequent line of code is ignored.

If you don't want to pass any value to the calling environment, just call the function with no arguments:

```
output();
```

In this case:

- No value is set to the SCRIPT RUNNER
- The ACTIONS (objects associated in output) are not executed

2.5 Inclusion of scripts

It is possible to include other SCRIPTs within a SCRIPT; this feature allows you to use previously created SCRIPTs as a sort of customized "library". To include a SCRIPT, simply call the same include function seen for libraries:

```
require("scriptName")
```

Where **scriptName** is the name of the SCRIPT to execute; for example:

```
require("trial")
```

2.6 Variable inspection

It is possible to inspect the value of variables and code objects through the following function:

```
printall
```

Example:

```
printall(variable)
```

This function outputs the contents of the variable or object; if structured, its content is returned on multiple lines, displayed in the DEBUG section of the runners page (see later chapters for details).

For text strings only, you can also use the system function for this purpose

```
print(string)
```

2.7 Tables

The *tables* are very flexible structures in the LUA language, which roughly correspond to the *array* of other languages; *tables* however can also have strings or non-numeric keys as indices, and therefore they are quite similar to *objects* of other programming languages (e.g. JavaScript).

Many LUA scripting libraries make use of tables to manipulate and return lists of items; the following syntaxes may be useful for working with them.

2.7.1 Number of elements

To know the number of elements of a *table* call the following function:

```
tablelength(<TABLE>)
```

2.7.2 Cycle Between Items

To cycle through the elements of a *table* use the following syntax:

```
for k,v in pairs(<TABLE>) do
    -- k is the key of element
    -- v is the element itself
end
```

For example, to scroll through a list of objects, filter them based on a certain type and interact with them, use the following code:

```
require("objects")

-- Searching for all lights in project
OBJECTS = objM.findObjs("Light","NAME",false)
for k,in in pairs(OBJECTS) do
    -- Considering only KNX lights
    if(v.TYPE == "EIBOBJECT") then
        -- Printing a debug message with info about current object
        -- k is the ID of object, v is the object itself
        system.debug("Light with ID " .. k .. " has name " . v.DB.NAME)
    end
end
```

For more information, refer to the subsequent chapters dedicated to object management libraries.

3 Scripts

3.1 Introduction

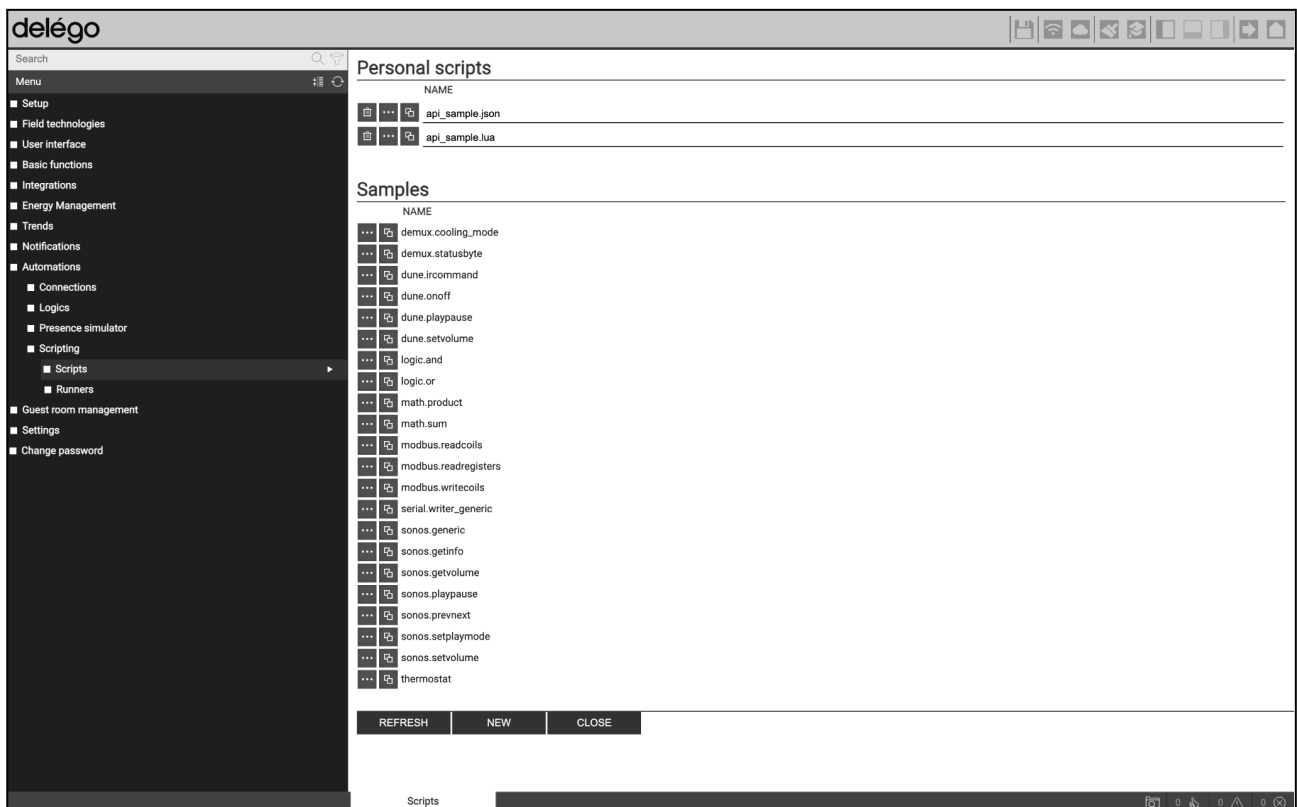
This chapter explains how to create, edit, delete and export custom scripts on DELÉGO SERVER. The information in this chapter applies to both PHP and LUA scripting, with the exception of any instructions or examples specific to the PHP language.

3.2 List

To access the scripts management page, simply:

- Log in to DELÉGO SERVER with a user with administrative permissions
- Access the CONFIGURATION environment
- Select AUTOMATIONS and then SCRIPTING from the menu
- Select "SCRIPTS"

A page similar to the following example is shown:



The top part of the page contains a series of example scripts preloaded in DELÉGO SERVER; they cannot be deleted or modified, but only consulted or duplicated (to create your own script starting from them).

The lower part instead shows the scripts present on the DELÉGO SERVER disk. To create a new script, simply press the "NEW" button at the bottom of the page: you are asked to indicate the

name to be assigned to the new script and, after a few seconds, you will receive confirmation that the new file has been created.

If you want to create a script starting from an existing example, simply press the corresponding "CLONE" button; also in this case you are asked to specify a name to assign to the new script.

To delete a script, press the corresponding "DELETE" (red) button; once confirmation is given, the file is removed from the DELÉGO SERVER disk (irreversibly).

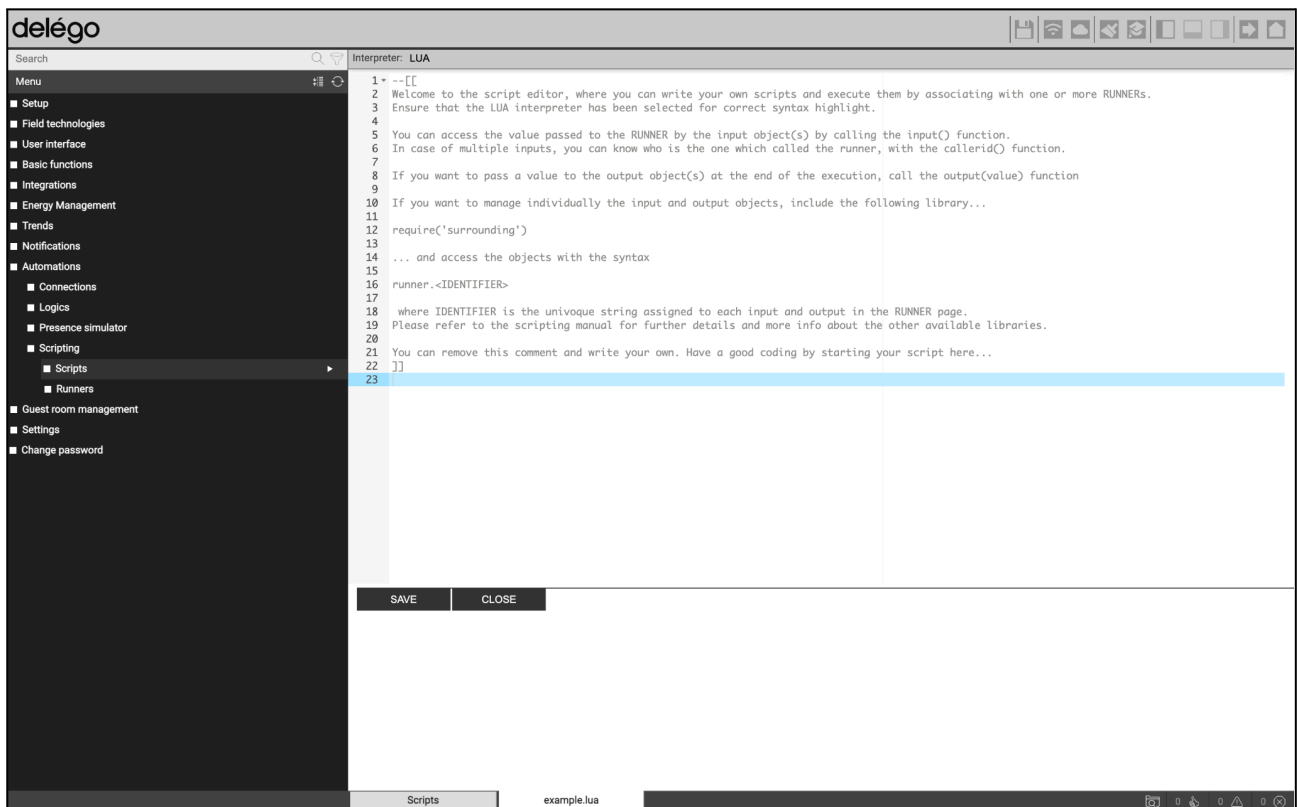
Finally, to access the consultation and/or modification form of a script (or an example), use the "EDIT" (green) button.

3.3 Editor

By selecting the EDIT button of a script or an example, a page containing the code that makes up the script itself is shown; in the case of the examples, the code cannot be modified.

In the case of a new script, the code has some comments which provide some initial indications on how to create the script, as described in more detail in this manual.

The following figure shows the edit page of a new script:



For further information on how to create a script, in particular by exploiting the libraries available, please refer to the following chapters.

Once the changes to the code have been completed, press the "SAVE" button to make them persistent; vice versa, pressing the "CLOSE" button before saving, any changes will be lost.

3.4 Examples

By way of example, in the continuation of this manual, unless otherwise indicated, the following example code will be used, which simply returns the value received as an INPUT VALUE, after reporting the same value in a DEBUG message (visible only during the execution simulation, as highlighted below).

```
value = input()
system.debug("Received as input the value " .. value)
output(value)
```

4 Runners

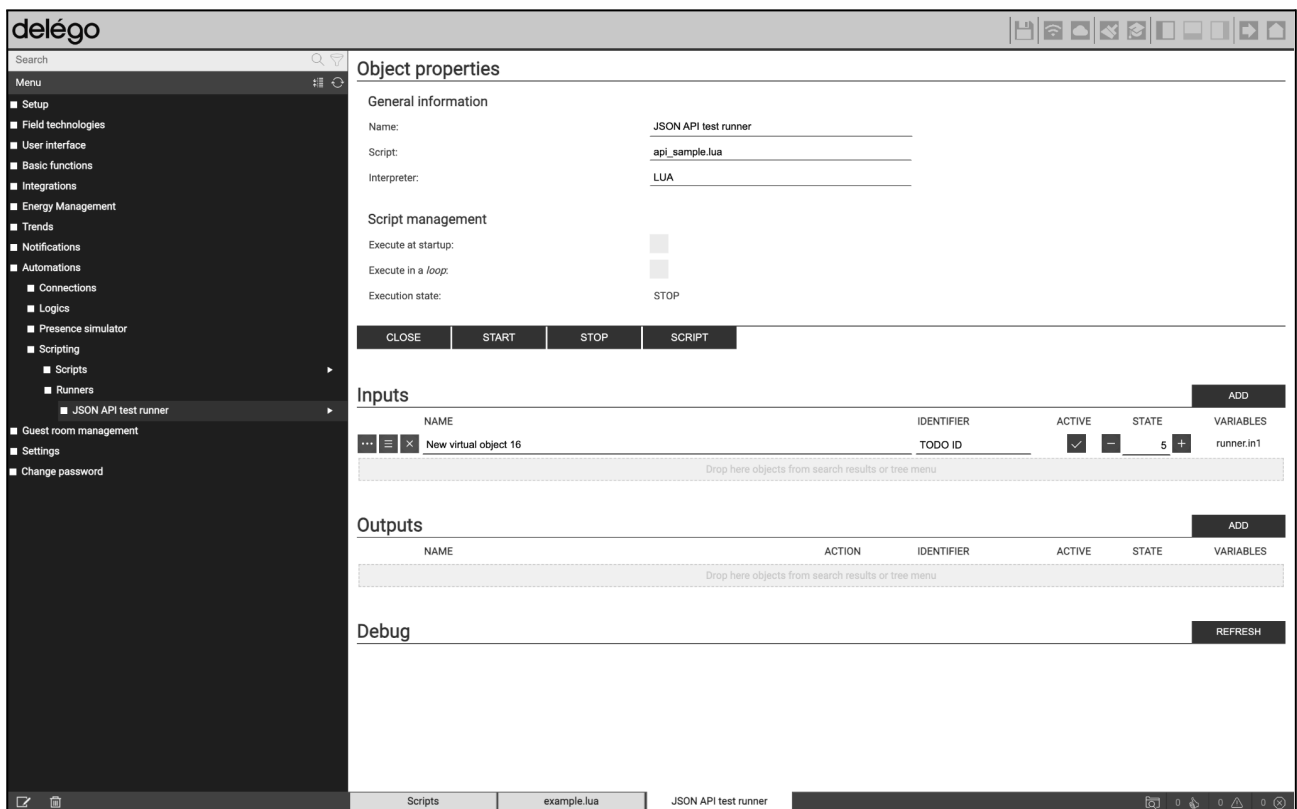
4.1 Introduction

This chapter illustrates how to create and manage objects of the SCRIPT RUNNER type, which allow you to execute SCRIPTS within the supervision of DELÉGO SERVER. The information in this chapter applies to both PHP and LUA scripting, with the exception of any instructions or examples specific to the PHP language.

4.2 Adding a new runner

To create a new SCRIPT RUNNER simply:

- Log in to DELÉGO SERVER with an administrative user, and access the configuration section
- Locate the "SCRIPTING" item under "AUTOMATIONS", then select "SCRIPT RUNNERS"
- Press the "ADD" button in the toolbar below
- Highlight the new created object and press the "dots" on the side, or - alternatively - the "EDIT" button in the toolbar at the bottom, to access its card



In addition to assigning a name to the SCRIPT RUNNER, it is also necessary to select which script will be executed, choosing it in the appropriate drop-down menu.

It is also possible to specify the following options for the SCRIPT RUNNER:

EXECUTE AT STARTUP	By selecting this flag, the SCRIPT RUNNER is automatically executed when the webserver starts
EXECUTE IN A LOOP	By selecting this flag, the SCRIPT RUNNER is executed iteratively for an indefinite number of times, until it is stopped manually or via an EVENT. For further details, please refer to the following "LOOP" section
LOOP TIME	If running in a loop, specify the duration in milliseconds of each iteration.

It is also possible to run or stop a SCRIPT RUNNER using the appropriate "START" and "STOP" buttons.

4.2.1 Inputs

The section **INPUTS** allows you to drag one or more objects which, when the status changes, determine the execution of the SCRIPT RUNNER.

Once you've dragged an item into this list, you need to decide which one **VALUE** must be passed to the SCRIPT RUNNER whenever the object itself changes state.

Unlike other parts of the software where this choice is guided through special drop-down menus, in this case the value must be specified manually. This field (visible only if you activate the ADVANCED view via the toolbar at the top right) can contain the following values:

\$VAL	The current value of the object is passed to the SCRIPT RUNNER as an input value. Default option.
\$NVAL	The value <u>inverted</u> of the object is passed to the SCRIPT RUNNER as an input value (it makes sense in case of boolean values)
Any numeric value or string	The value is passed to the SCRIPT RUNNER as an input value, regardless of the state of the object itself

In any case, the value is available within the script through the call

```
input()
```

already described above. Also, in the case of scripts associated with more than one input object, it is possible to obtain the ID of the object that has changed (and which is passing the input value) through the function call

```
callerid()
```



The value indicated in this field is not the only way to access, within the SCRIPT, the value of the related object as an input. Through the **surrounding** library in fact, as better described below, it is possible to know the status of all the inputs and carry out operations on them; for further details, refer to the chapter dedicated to this library.

4.2.2 Outputs

Similarly, dragging objects within the section **OUTPUTS** it is possible to determine its execution at the end of the SCRIPT.

For each object it is necessary to specify which action to execute and (if the object foresees it) which value to pass at the end of the execution of the SCRIPT RUNNER; in the latter case, choose STATE to pass to the object the value returned in output from the SCRIPT with the command

```
output(...)
```



As previously highlighted, it is possible to inhibit the automatic execution of the associated objects as outputs at the end of the SCRIPT, by not passing any value to the function **output()**.

In this case, the possible command of the associated objects as outputs is entrusted to the SCRIPT code itself, as better detailed in the chapter dedicated to the library "surrounding".

Also for the outputs it is possible to associate an **identifier** and display the name of the variable with which to refer to the output in the SCRIPT.

4.3 Loop

A script can be run continuously in the "background" by activating the *loop* option in the SCRIPT RUNNER tab. Typically, when you enable this option, you also enable the *run at startup* option, so that the code is re-executed, if the system reboots.

4.3.1 Cyclic Execution

In *loop* mode, the following function is cyclically called:

```
main_loop()
```

which you need to define inside the script, for example:

```
function main_loop()  -- Main program going into a cyclical loop
    system.debug("Cyclic Execution" )
end
```

This function is called periodically with a waiting time, between one execution and the next, to be set in the *loop time* option (specify a value expressed in milliseconds); the lower the time set, the higher the reactivity of the RUNNER, but also the load on the CPU of the webserver.

4.3.2 Execution At Interrupt

It is possible to define another function in the script, which is called every time one of the inputs changes state: for example...

```
function event_loop(event)  -- Function called when the status of an input changes
    printall(event)
end
```

The function argument is a *table* containing the following information:

caller	ID of the incoming object that has changed state
value	Value received from the object
hashcode	Any PIN passed by the subject Alternatively the string "NO-HASHCODE" is passed
optionals	Concatenation of attributes of the relationship between the object in question and the RUNNER If an IDENTIFIER has been specified for the input object, it is supplied through the "tag_complex_element" attribute

Example of the output of the *printall* on the function argument:

```
(TABLE)
__[optionals]= tag_complex_element='CUCINA'
__[hashcode]= NO-HASHCODE
__[value]= 1
__[caller]= 123
```

Where

123	ID of the calling object
1	Value received
KITCHEN	Identifier associated with the object in the RUNNER card

This function can be better exploited by including the SURROUNDING library presented later in this manual, which provides methods for interacting more comfortably with the INPUTS of the RUNNER.

4.4 Debugging

It is possible to insert within the scripts DEBUG messages (aimed at monitoring the activities carried out by the script under construction), WARN (useful for notifying the execution environment of warning messages) and ERROR messages (necessary to signal and trace any anomalies during the execution of the script that compromise its correct completion).

To insert these messages it is sufficient to call, respectively, one of the following functions:

```
system.debug()  
system.info()  
system.warn()  
system.err()
```

Example:

```
system.debug("This is a debug message")  
system.info("This is an info message")  
system.warn("This is a warning message")  
system.err("This is an error message")
```

These messages are displayed in the DEBUG section, colored differently according to the level.

5 Models

5.1 Introduction

Starting from release 2.8 it is possible to define a MODEL for a script, by specifying in advance the list of *inputs and outputs* that the runners (associated to that script) will have. This leads to two main advantages:

- In the edit page of a runner, when associating an object as input or output, the *identifier* can be selected from a drop-down menu instead of entering it manually
- It is possible to place the runner in a CONNECTION or a LOGIC TASK as any other object

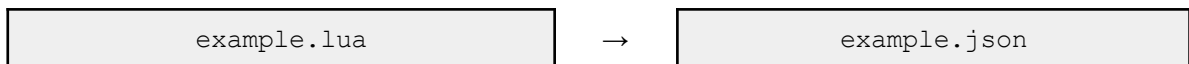
In other words, the MODELS permit to extend the functionality of the webserver by defining new blocks that can be fully integrated in the graphic editor for the automations, with the flexibility to write the code by your own.

The MODELS can be exported along with their referred SCRIPTS and reused in other projects; in this way, it is also possible to share them to other users.

5.2 Creating a model

To create a new MODEL for a SCRIPT, do the following steps:

- Enter the SCRIPTS list and press the button to ADD a new one
- The MODEL name must be exactly the same as the referred SCRIPT,
- The extension must be **.json** instead of **.lua**:



When entering the editor, the following example is displayed:

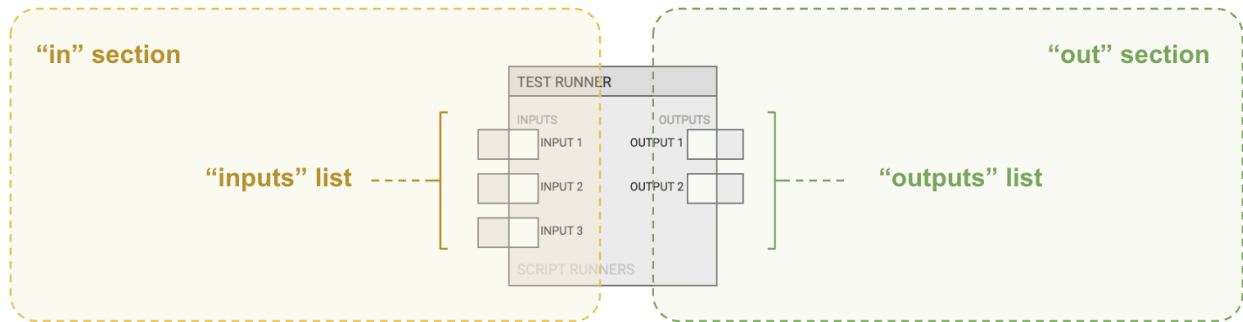
```
{
  "in":{
    "inputs":{
      "label":"INPUTS",
      "nodes":{
        "in1":"INPUT 1",
        "in2":"INPUT 2",
        "in3":"INPUT 3"
      }
    }
  },
  "out":{
    "outputs":{
      "label":"OUTPUTS",
      "nodes":{
        "out1":"OUTPUT 1",
        "out2":"OUTPUT 2"
      }
    }
  }
}
```

```

    }
  }
}

```

In the moment a RUNNER, associated with this SCRIPT, is dropped into an automation, it would appear this way:



The JSON structure is organized in two SECTIONS, one for the *input* nodes (in), the other for the *outputs* (out).

Each SECTION can contain one or more LISTS of nodes; each list has a label displayed on top of the contained NODES. In the following example, of an hypothetical HVAC appliance, more lists have been added on both the IN and OUT sections:

```

{
  "in":{
    "setpoints":{
      "label":"SETPOINTS",
      "nodes":{
        "set1":"SET 1",
        "set2":"SET 2",
        "set3":"SET 3"
      }
    },
    "sensors":{
      "label":"SENSORS",
      "nodes":{
        "sens1":"SENSOR 1",
        "sens2":"SENSOR 2",
        "sens3":"SENSOR 3",
        "sens4":"SENSOR 4",
      }
    },
    "enables":{
      "label":"ENABLES",
      "nodes":{
        "en1":"EN 1",

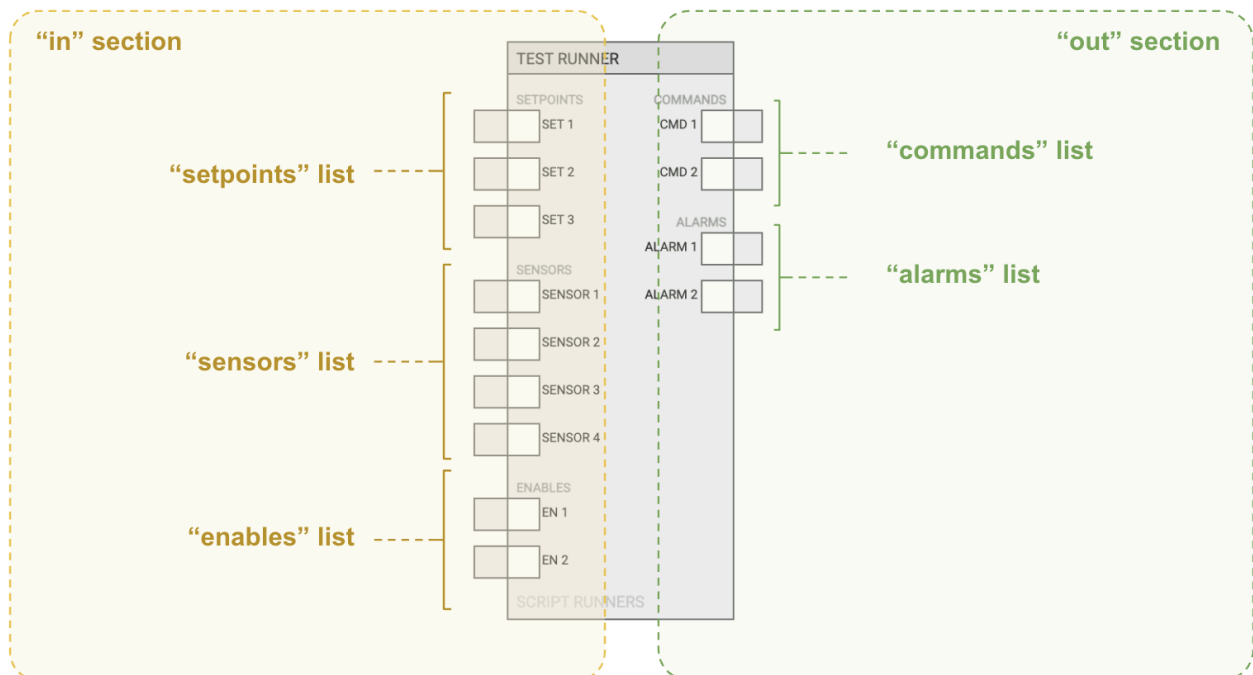
```

```

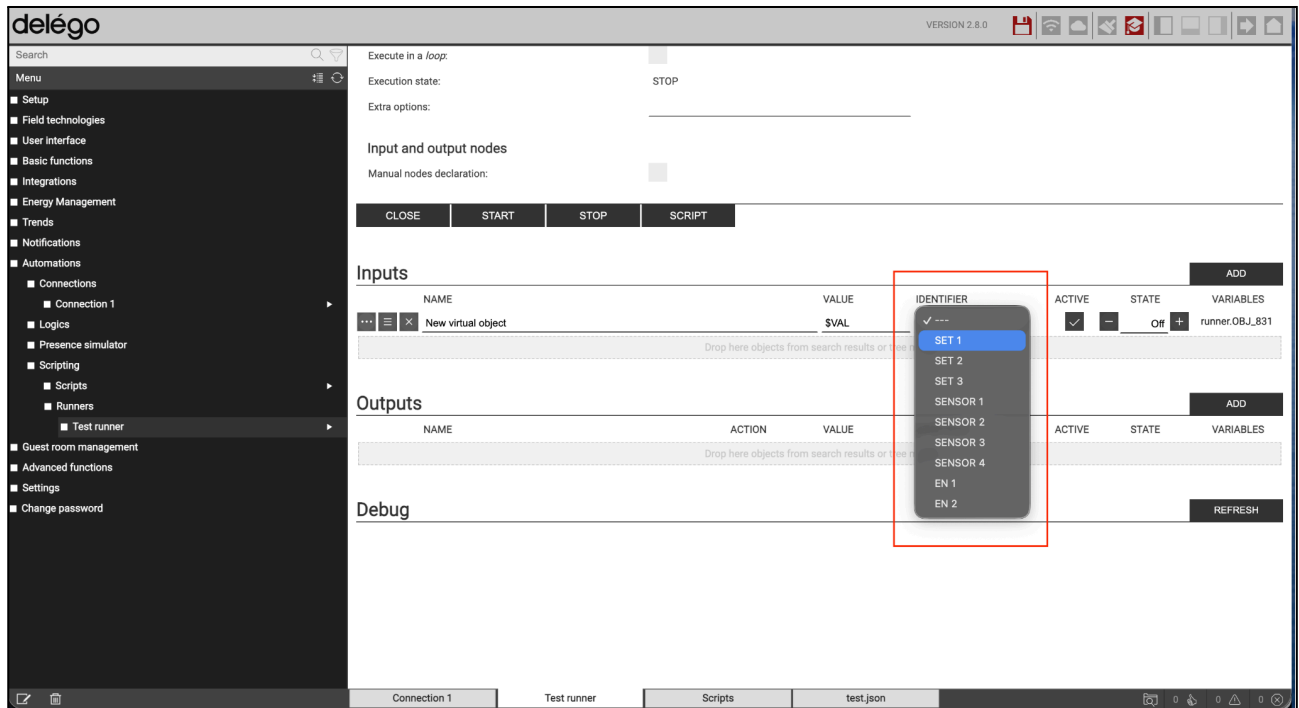
        "en2": "EN 2"
    }
},
"out": {
    "commands": {
        "label": "COMMANDS",
        "nodes": {
            "cmd1": "CMD 1",
            "cmd2": "CMD 2"
        }
    },
    "alarms": {
        "label": "ALARMS",
        "nodes": {
            "al1": "ALARM 1",
            "al2": "ALARM 2"
        }
    }
}
}

```

The result in the connection editor would be the following:



In the edit page of the RUNNER(s) associated to the script, the identifier of the inputs and output can be selected from a list that uses the definition contained in the JSON, as you can see in the following sample screenshot:



It is possible to manually override the MODEL associated to a SCRIPT for a specific runner (or adding a nodes declaration if missing the model). In the edit page of the runner, enable the



MANUAL NODES DECLARATION

Flag and enter the desired JSON in the textarea that appears.

6 Objects Library

6.1 Introduction

The "objects" LUA library provides a class with a set of methods for interacting with the objects of the supervision project.

This library is the basis of the "SURROUNDING" library, which allows easy interaction with the INPUTS and OUTPUTS of the RUNNER, subject of the next chapter of this manual.

6.2 Inclusion

To include the library in a SCRIPT it is necessary to insert the following line at the beginning:

```
require("objects")
```

The inclusion of the library provides a class called "object manager" accessible via the keyword

```
objM
```

whose methods are described below.

6.3 Loading Objects

To load a single supervision object into a LUA variable it is necessary to call the following method:

```
<OBJ> = objM.getObj(XXX,<ADD TO COLLECTION>)
```

where XXX is the numeric ID of the object to be loaded, and <OBJ> is any variable name to assign the object to.

The optional second argument, which is expected to be a boolean, determines if the object returned by the function, is added to the internal collection of objects; this is useful to optimize the following usage of the same object. The default is **true**.

Example:

```
KITCHEN_LIGHT= objM.getObj(123)
```

If the object has already been called before, this method gives you an *instance*; otherwise, a query is made to the project database, and in addition to assigning it to the variable, it is saved in a *collection* internal to the script, for subsequent queries.

If you want to load multiple objects simultaneously, you can call the method

```
<ARRAY> = objM.getObjs(<array of IDs>)
```

which must be passed an array (*table*) of IDs of as many objects, for example:

```
OBJECTS = objM.getObjs({123,456,789})
```

The function returns an array containing the requested objects, in which the index is represented by the ID; in the previous example it is therefore possible to subsequently access a single object of the collection through the syntax

```
KITCHEN_LIGHT = OBJECTS[123]
```

The objects obtained by multiple loading, as an alternative method, can be subsequently recovered through the single loading method seen previously...

```
KITCHEN_LIGHT =objM.getObj(123)
```

... without further readings from the project database (being already in the *collection* internal to the script).

6.4 Object Properties

The objects loaded with the functions seen previously make available a series of primary attributes, shown in the following table.

The variables that represent objects of the supervision project have the following properties, which can be used in the SCRIPT (pay attention to case sensitive which must be respected for correct syntax):

PROPERTY	DESCRIPTION	EXAMPLE
ID	Unique ID of the object	<OBJ>.ID
TYPE	"Functional" type of object, used by communication services. Some examples: KNX object: "EIBOBJECT" Variable: "VIRTUALOBJECT" Scene: "SCENARIO" Script runners: "CSCMD" <i>NOTE: For a complete list of types, see the Appendix</i>	<OBJ>.TYPE
PHPCCLASS	Graphic type of object, used by the web interface. Some examples: KNX objects: "dpadEibObject" Variables: "dpadVirtualObject" Scenarios: "dpadScenario" Script runners: "dpadScriptRunner"	<OBJ>.PHPCCLASS

	<i>NOTE: there is not always a unique correspondence between "type" and "phpclass": in some cases, more than one "phpclass" share the same "type". For a complete list, please refer to the Appendix</i>	
VALUES_TYPE	Data type associated with the object. The encoding of this field depends on the TYPE.	<OBJ>.VALUES_TYPE
DB	Collection of all database fields related to the object in question. See below for details	<OBJ>.DB
status	Current state of the object. NOTE: this property has meaning only for scenarios; in this case: 1 → stopped scenario 2 → running scenario	<OBJ>.status
value	Current value of the object NOTE: this property has meaning for all object types that have a status, scenarios excluded	<OBJ>.value

The DB collection contains the list of all the database fields relating to the object; they can be read via syntax

```
<OBJ>.DB.<FIELD>
```

for example:

```
KITCHEN_LIGHT =objM.getObj(123)
LIGHT_NAME =KITCHEN_LIGHT.DB.NAME          -- Returns for example "Kitchen Light"
```

Below is a list of the main fields:

FIELD	DESCRIPTION
NAME	Object name
DESCRIPTION	Item description
MSP	Main attribute of the object; the meaning of this field depends on the TYPE, usually it contains the <i>address</i> (e.g. KNX group address)
UNIT	Where available, it represents the unit of measure
IS_WRITABLE IS_READABLE	Permission flag respectively for command and object reading

IS_VISIBLE	Object visibility flag
IS_STOPPABLE	Object STOP permission flag (valid only for SCENARIOS)
IS_BOOLEAN	Flag that establishes whether the object is of type BOOLEAN (that is, it only admits values 1 or 0) or not
IS_PROTECTED	Object protection flag against changes made by the user interface
ENABLE_FLAG	General enable flag of the object for the operation and interaction with other objects
TRACK_FLAG	Object tracking flag in the logging database

Finally, a sub-collection of *options* is accessible with the following syntax

```
<OBJ>.DB.options.<FIELD>
```

for example:

```
KITCHEN_LIGHT =objM.getObj(123)
CATEGORY =KITCHEN_LIGHT.DB.options.category    -- Returns for example "lighting"
```

For a complete breakdown of all fields, it is recommended to inspect the object via the method *printall* documented above, as the list of available options and their values strongly depend on the type of object.

6.5 Commands

It is possible to send a command to an object, setting a new value, through the method

```
<OBJ>:set(<VALUE>,doNow)
```

Where

<VALUE> New value passed to the object

doNow Boolean, if TRUE the command is sent immediately (recommended)

 If FALSE the command is executed at the next iteration (only for runners in loop)

For example:

```
KITCHEN_LIGHT =objM.getObj(123)
KITCHEN_LIGHT:set(1,true)    -- Turns on the light immediately
```

6.6 Search For Objects

It is possible to search for one or more objects in the project starting from one of the fields in the database, listed above; to get a single object use the method

```
<OBJ> = objM.findObj(<KEY>,<FIELD>,<EXACT_MATCH>)
```

Where

<KEY>	Keyword, to be specified as a string
<FIELD>	Name of the database field to search
<EXACT_MATCH>	Flag that specifies whether the keyword must exactly match the database field, or whether a part of the database value is sufficient to match the keyword.

Specify this flag in boolean format *true* or *false*

Example:

```
KITCHEN_LIGHT = objM.findObj("Kitchen Light","NAME",true)
```

Also in the case of the search, it is possible to request the loading of more objects that meet the search criterion, through the method

```
<ARRAY> = objM.findObjs(<KEYS>,<FIELD>,<EXACT_MATCH>)
```

The first <KEYS> argument can be a single keyword (in which case it must be passed as a string) or an array of keywords; in the second case, based on the <EXACT_MATCH> field, the following types of searches are performed in the database respectively:

<EXACT_MATCH> = true	All the objects that have exactly in the field <FIELD> the values contained in the array <KEYS> are loaded
<EXACT_MATCH> = false	All objects are searched which contain in the field <FIELD> at least one of the keywords specified in <KEYS>

Examples:

```
-- Search for all objects that contain a keyword in the name
```

```
<ARRAY> = objM.findObjs("Light","NAME",false)
```

```
-- Search for a specific list of objects by name
```

```
<ARRAY> = objM.findObjs({"Kitchen light","Living room light"},"NAME",true)
```

6.7 Search by address

In order to search for objects with their address (e.g. the KNX group address) it is possible to use the following *wrapper* of the generic **findObjs** function:

```
<OBJ> = objM.getObjByAddr(<ADDR>,<ADD TO COLLECTION>)
```

Example:

```
myObj = objM.getObjByAddr('1/0/1')
```

6.8 Editing objects

It is possible to programmatically modify the value of a field in the database relating to an object, through the method

```
<RESULT> = objM.updatFieldObj(<ID>,<FIELD>,<VALUE>)
```

Where

<ID>	ID of the object to edit
<FIELD>	Name of the database field to edit
<VALUE>	New value to assign to the field. Pass this argument as a string.

The return value of the method informs about the successful modification (*true*) or with error (*false*).

7 Surrounding Library

7.1 Introduction

The LUA "SURROUNDING" library allows you to easily interact with the INPUTS and OUTPUTS of the RUNNER, extending and specializing the previously seen methods of the "OBJECTS" library.

7.2 Inclusion

To include the library in a SCRIPT it is necessary to insert the following line at the beginning:

```
require("surrounding")
```

Starting from the line immediately following the inclusion of the library, the following object is available to the SCRIPT:

```
runner
```

This object represents the SCRIPT RUNNER that is executing the SCRIPT, and allows you to interact with it within the SCRIPT.

7.3 Inputs and Outputs

The **runner** object also makes available its possible relationships with the INPUTS and OUTPUTS configured in the SCRIPT RUNNER form through the following arrays:

```
runner.INPUTS    //Inputs  
runner.OUTPUTS   //Outputs
```

Within the arrays, the different objects are represented by the variable "OBJ_XXX" where XXX is the ID of the object, for example:

```
runner.INPUTS.OBJ_123    //Incoming object with ID 123
```

These objects provide the properties and methods described in the previous chapter.

Additionally, they provide a *collection* containing information about the RELATIONSHIP with the RUNNER:

PROPERTY	DESCRIPTION	EXAMPLE
----------	-------------	---------

RELATION	Collection of attributes of the relationship with the RUNNER	<OBJ>.RELATION
----------	--	----------------

The report fields are listed in the following table:

FIELD	DESCRIPTION		
ACTION	Action performed on the target object of the relationship		
ACTION_VALUE	Value passed to the target object		
options	Collection of optional relationship values, including: <table border="1" data-bbox="395 651 1433 703"> <tr> <td>tag_complex_element</td><td>ID assigned to the ENTRY / EXIT</td></tr> </table>	tag_complex_element	ID assigned to the ENTRY / EXIT
tag_complex_element	ID assigned to the ENTRY / EXIT		

7.4 Identifier

It is possible to associate to each object an **identifier** (which must be a string without spaces or special characters) which facilitates its identification within the SCRIPT, when using the SURROUNDING library. For example, if the identifier "TEST" is associated with an object, it is available in the code by writing

```
runner.TEST
```

To know the ID of an INPUT or OUTPUT object to a RUNNER known its ID, it is possible to call the following functions:

<code>runner.getIdentifierById(id)</code>	-- Identification search between INPUTS and OUTPUTS
<code>runner.getInputIdentifierById(id)</code>	-- Identification search only among the INPUTS
<code>runner.getOutputIdentifierById(id)</code>	-- Identification search only among the OUTPUTS

For example, inside a RUNNER in *loop* it is possible to intercept the change of state of an input, identify it and manage it with the following example code:

```
function event_loop(event)
    local IDENTIFIER = runner.getInputIdentifierById(event.caller)
    system.debug("Input " .. IDENTIFIER .. " changed to value " .. tostring(event.value))

    local OBJ = objM.getObj(event.caller)
    system.debug("Object name: " .. OBJ.NAME)
end
```

8 Messages library

8.1 Introduction

The "messages" LUA library allows you to send video and email notifications directly within a SCRIPT, without necessarily having to create objects within the supervision project.

8.2 Inclusion

To include the library in a SCRIPT, simply insert the following line at the beginning:

```
require("messages")
```

Starting from the immediately following line, the following class is available to the SCRIPT:

```
msgM
```

This class ("message manager") provides a series of functions (or "methods") for loading supervision objects and performing operations on them.

8.3 Video Notifications

To send a new video notification "on the fly" while a script is running, simply call the function

```
msgM.sendVideoMsg(msg, level)
```

Where

msg	Message to send
level	Message priority level. Possible values:
0	Alarm (default)
1	Warning
2	Information

Example:

```
msgM.sendVideoMsg("Received input value " .. input() .. ", 2);
```

The function returns 0 if successful, an error code otherwise.

8.4 Mail Notifications

Similarly, an e-mail message can be sent "on the fly" through the call

```
msgM.sendEmailMsg(msg, to, subject, cc, bcc)
```

Where

msg	Message to send
to	Email address(es) to send the message to In case of multiple addresses, separate them with ";"
subject	Message subject [Optional]
cc	Email address(es) in copy [Optional]
bcc	Blind copy email address(es) [Optional]

The function returns 0 if successful, an error code otherwise.

9 JSON library

9.1 Introduction

This library allows you to manipulate strings in JSON format by transforming them into data structures of *metatable* at LUA.

9.2 Inclusion

To include the library in a SCRIPT, simply insert the following line at the beginning:

```
JSON = require("json")
```

9.3 Decoding

Once the library is included and assigned to the JSON class in the code, it is possible to decode a string in JSON format through the method

```
<TABLE> = JSON.decode(<JSON_STRING>)
```

Example:

```
JSON.decode(' [1,2,3,{"x":10}] ') -- Returns { 1, 2, 3, { x = 10 } }
```

9.4 Encoding

Similarly, it is possible starting from a *table* LUA structure to get a JSON representation inside a string with the method

```
<JSON_STRING> = JSON.encode(<TABLE>)
```

Example:

```
JSON.encode({ 1, 2, 3, { x = 10 } }) -- Returns '[1,2,3,{"x":10}]'
```

9.5 Credits

This library is an OPEN-SOURCE project not developed by EKINEX; for further details, please refer to:

<https://github.com/rxi/json.lua>

10 URL library

10.1 Introduction

This library allows you to call up a URL and handle the response. It can be used to invoke HTTP(S) API via GET calls, and decode the result (e.g. from JSON format).

10.2 Inclusion

To include the library in a SCRIPT, simply insert the following line at the beginning:

```
require("url")
```

The library provides a "URL manager" class that can be called with the keyword

```
urlM
```

10.3 Getting URLs

To retrieve a URL via GET, and assign the response to a variable, use the following syntax:

```
<RESPONSE> = urlM.get(<URL>)
```

Example:

```
JSON = require("json")
require("url")

-- Returns your public IP in JSON format
-- Is: {"ip": "150.60.100.80"}
response = urlM.get("http://ip.jsontest.com")
my_address = JSON.decode(response) ["ip"]
system.debug("My IP public address is " .. my_address)
```

11 Appendices

11.1 Functional Types

The following table shows a complete list of all the functional types of objects provided by DELÉGO SERVER, used by the communication services to carry out the "background" operations of the webserver. Usually we refer to this type of object as "TYPE".

For each of them, in addition to the type and description, the possible actions are reported; the latter can be called with the methods seen previously for the command of the objects.

TYPE (TYPE)	DESCRIPTION	ACTIONS
VIRTUALOBJECT	Variable	SETVALUE → Set a value to the object
SCENARIO	Scenario	EXECUTE → Execute the scenario STOPEXECUTION → Stop the execution
LOGIC	Logic	EVALUATE → Currency to logic
CONDITION	Condition	CALCULATE → Calculate the comparison of values
CLIENTBROWSER	Client	REDIRECT → Make a page jump on the client VOIPCALL → Make a VOIP call to the client's IP address
NOTIFY_VIDEO	Video notification	INSERT → Create a video message
NOTIFY_EMAIL	Email notification	SENDMAIL → Send an email message
INTEGRATOR	Supplement	INTEGRATE → Update the integrator calculation RESET → Resets the integrator value to 0
EIBOBJECT	KNX object	SETVALUE → Set a value on the corresponding KNX group address GETVALUE → Issues a request on the KNX bus to read the updated status
CSCMD	Script runner	RUN → Execute the script QUIT → End the execution of the script
USER	User	REDIRECT → Make a page jump on all clients where the user is connected VOIPCALL → Make a VOIP call to all customers where the user is connected
USERGROUP	User group	

USER PERMISSION	Permit	
CONTAINER	Composite object	
CAMERA	IP camera	
PBXELEMENT	IP phone IP video intercom VOIP call group	VOIPCALL → Make a VOIP call destined for the object

11.2 Web Types (Phpclass)

The following table shows a complete list of all object types expected by DELÉGO SERVER regarding the web interface. Typically, this type of object is referred to as "PHPCLASS".

For each of them the corresponding functional type (or "TYPE") is reported; it is possible for more than one PHPCLASS to refer to the same TYPE. If not present, the PHPCLASS has no functional counterpart for communication services; this typically happens for all web types that have a purely graphic function (environments, links, etc...)

TIPO (PHPCLASS)	DESCRIPTION	REFERENCE TYPE
dpadObject	generic object *	
dpadGroup	Environment (or "folder" in a general sense)	
dpadUrl	Connection	
dpadSysCmd	System command *	SYSCMD
dpadSysCmdWait	Scenario wait command	SYSCMD
dpadUser	User	USER
dpadUserGroup	User group	USERGROUP
dpadUserPermission	Access permission for user groups	USER PERMISSION
dpadTriggerObject	System trigger on DB change *	TRIGGEROBJECT
dpadContainer	Composite object	CONTAINER
dpadVirtualObject	Variable	VIRTUALOBJECT
dpadScenario	Scenario	SCENARIO

dpadPbxExtension	IP phone	PBXELEMENT
dpadPbxTrunk	VOIP telephone line	
dpadPbxQueue	VOIP call group	
dpadPbxDoorOpener	IP video intercom	
dpadLogic	Logic	LOGIC
dpadCondition	Condition	CONDITION
dpadClientBrowser	Client	CLIENTBROWSER
dpadNotifyVideo	Video notification	NOTIFY_VIDEO
dpadNotifyEmail	Email notification	NOTIFY_EMAIL
dpadIntegrator	supplement	INTEGRATOR
dpadEibObject	KNX object	EIBOBJECT
dpadCamera	IP camera	CAMERA
dpadScriptRunner	Script runner	CSCMD
dpadLoadControl	Load control	CSCMD

** : types of object not usually used or available in the web interface*

11 Revision history

The following table contains the most relevant changes in the software releases, concerning the topics of this application manual:

Release	Changes	Date
2.8.0	Introduced the MODELS for the definition of the input and output nodes of the runners associated to a script	15/04/2026
2.7.0	Added JSON library	01/09/2025
2.6.0	First revision of the document	02/04/2025



Before the software release 2.6 the application manuals of DELÉGO SERVER were organized differently. An exact correspondence with older versions of the manuals and the changelog could not be found for this reason.