Scripting Module Manual

Version 2.0
REV02-20160415

GENERAL INFORMATION

DIVUS GmbH
Pillhof 51
I-39057 Eppan (BZ) - Italy

You are always welcome to inform us of errors or make suggestions for improving the program.

The present agreement also applies to special appendices to the manual.

This manual can contain terms and descriptions, which improper use by third can harm the copyrights of the author.

Please read the manual before beginning and keep the manual for later use.

The manual has been conceived and written for users who are experienced in the use of PCs and automation technology.

CONVENTIONS

[KEYS]	Keys that are to be pressed by the user are given in square brackets, e.g. [CTRL] or [DEL]
`COURIER`	On-screen messages are given in the Courier font, e.g. `C:\>`
`COURIER BOLD`	Keyboard input to be made by the user are given in Courier bold, e.g. `C:\>`**`DIR`**
„…“	Names of buttons to be pressed, menus or other onscreen elements and product names are given within double quotes. (e.g. “Configuration”).
PICTOGRAMS	In this manual the following symbols are used to indicate particular text blocks.
	*Caution!* A dangerous situation may arise that may cause damage to material.
	*Hint* Hints and additional notes
	*New* New features

INDEX:

1 Introduction

1.1 WHAT IS KNXCONTROL

KNXCONTROL is a product family for the supervision and visualisation of home & building automation systems, which have been realized on the basis of the worldwide KNX standard. The KNXCONTROL devices allow managing of all functions of the system through browser access from any PC / MAC / Touchpanel, tablet and smartphone of the last generation, within the same network or remotely via internet.

Through the home page www.divus.eu all datasheets, brochures and technical manuals necessary for the configuration and usage of the KNXCONTROL products can be downloaded for free.

The software to manage and visualize such a system is *DIVUS OPTIMA*. *OPTIMA* offers a complete set of basic functionalities which may be further expanded through a modular system.

1.2 WHAT IS THIS MANUAL

This manual explains the functionality of the scripting module of the KNXCONTROL devices, which can be used to extend their basic functionalities. The following list shows some application examples:

- Interaction with the objects of the software (read values, evaluate actions, send commands)
- Sending customized notifications
- Interaction with the operating system
- Sending / Receiving commands through LAN and serial interface

The manual will guide you step by step through the creation of a customized script and its execution within the visualisation of the KNXCONTROL product. Furthermore you will find a detailed explanation of the different pre-installed sample scripts, which will give you a good overview of the possibilities of the scripting module.

1.3 REQUIREMENTS

In order to be able to use the scripting functionalities of the KNXCONTROL devices in the best way, you should:

- have a KNXCONTROL device
- have good knowledge using OPTIMA
- have good knowledge of ETS as well as of the KNX technology in general
- have basic knowledge of the scripting language PHP

2 General Overview

2.1 SCRIPTS

The SCRIPTS are program files containing PHP code, which are directly stored on the flash memory of the KNXCONTROL device and can be accessed through the configuration area (*ADMINISTRATION*) of OPTIMA. These files possess an own configuration window, which permits to create, edit, delete and also export the script files in order to use them on other devices, too.

The script files are kept even after a reset of the database of the KNXCONTROL device. Since they are not part of the database-backup, if necessary, they have to be ported from one device to another manually.

The scripts must be created according to the instructions and directives within this manual, by using the offered libraries. Each kind of PHP code not fulfilling the defined guidelines can compromise the correct functionality of the KNXCONTROL device.

Your KNXCONTROL device contains already a few script examples, which give you an overview of the built-in functions and methods. The direct editing of those scripts is not allowed, but you can clone them and then work on the created copies.

2.2 RUN - SCRIPTS

After creating a new script (either by copying a sample script or by creating a new one) it must be connected to one or more "run-scripts"; those special objects will execute the code contained in the script file whenever:

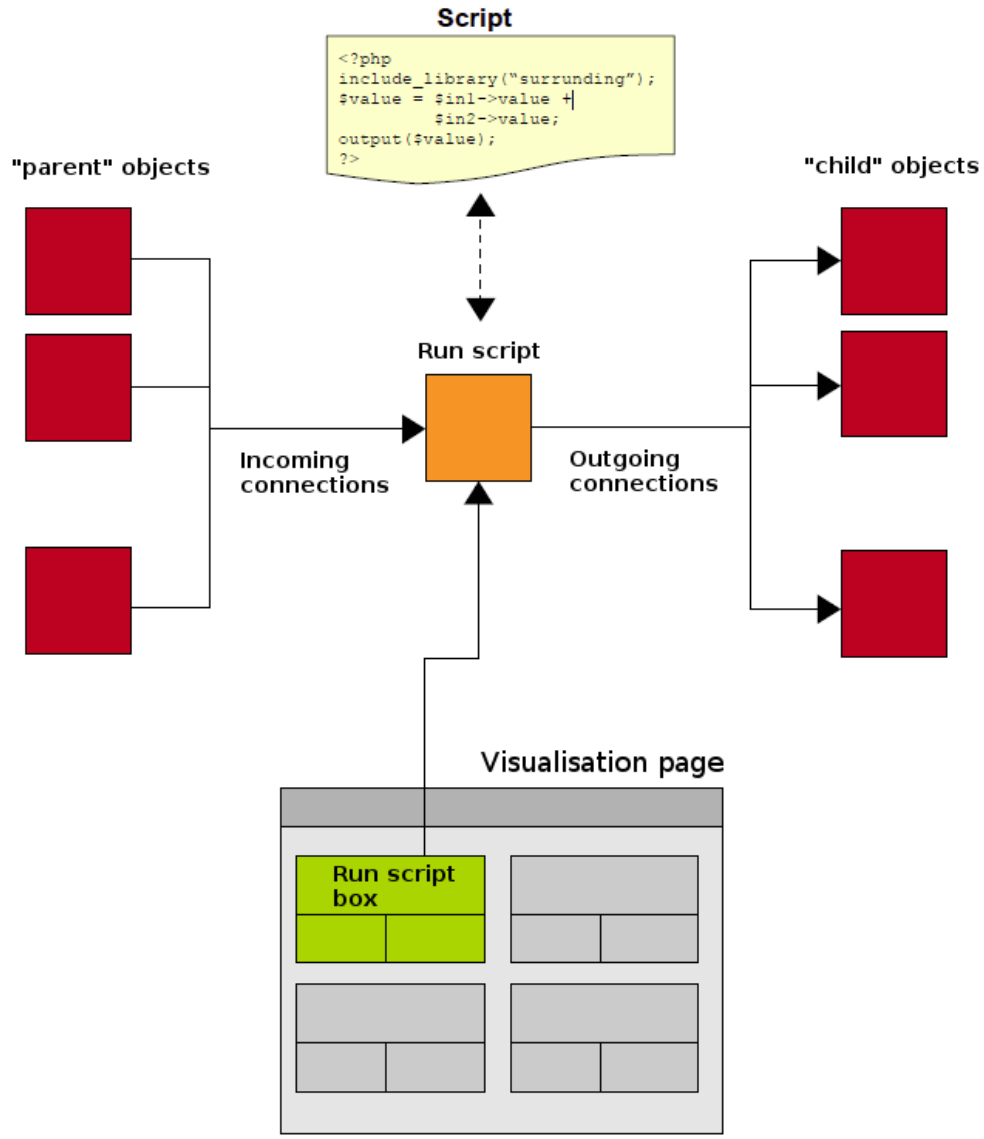
- the run-script is started manually through its "box" from within the visualisation pages
- the value of an object, which has been defined as INPUT (Incoming connection) of the run-script, is changed. This behaviour is similar to all other objects inside OPTIMA.

The SCRIPTS themselves cannot be executed directly; it will always be necessary to start them through a connected RUN-SCRIPT, which can be handled like any other object of the software: it has an own icon and a name, it can be assigned to rooms or function pages, it can be connected with outgoing and incoming connections (passive or active events), it can be connected to a scenario etc.

The advantages of such a structure divided in SCRIPTS and RUN-SCRIPTS are:

- the possibility to assign the SCRIPTS to more than one RUN-SCRIPT, in order to control/evaluate/ command different objects using the same functionality of the script.
- the possibility to export the SCRIPTS independently from their usage in the current project, in order to use them also on other devices.

The following scheme shall give you an idea about the information flow between RUN-SCRIPT and the connected objects / events as well as the data flow when commanding the RUN-SCRIPT:



2.3 LIBRARIES

Optima offers a set of programming libraries which provide different functions for the interaction with the connected objects. The libraries are not automatically accessible from the scripts, but must be included explicitly using the following command:

```
include_library(<Name of the library>);
```

Example:

```
include_library("serial");
```

Since the inclusion of libraries costs both work and time during the execution of the script, it is recommended to include only the libraries that are really needed / used inside the script. It is also possible that libraries depend on other libraries; in this case the depending libraries must not be included separately, since the main library will include them automatically.

The following table shows an overview of the available libraries as well as their dependencies. The single libraries will be explained in the further chapters of the manual:

LIBRARY	DESCRIPTION	DEPENDENCY
object	Permits to interact with the different objects of the visualisation (reading current values, changing properties, sending commands)	db
surrounding	Permits to interact directly with the connected run-script and the passive / active events	db object
serial	Permits to read / write strings through the RS232 interface of the KNX-CONTROL device (or through compatible USB-RS232 converters)	
modbus	Permits to manage one or more MODBUS SLAVE devices through TCP / UDP (reading and writing values)	
sonos	Allows to control the basic functions of a Sonos device (multi-room capable speaker systems) connected to the same network as the KNX-CONTROL device.	
dune	Allows to control Dune-HD devices (multi-room speaker systems)	
messages	Allows to access and manage notifications – both on-screen and via email	

2.4 INPUTS AND OUTPUTS

If a RUN-SCRIPT is executed, it will always have an INPUT value:

- If the RUN-SCRIPT is started manually through a page of the visualisation, the value depends on the user (e.g.: input of a value through a text field, pressing a button, etc. - depends on the graphical design assigned to the RUN-SCRIPT)
- If the RUN-SCRIPT is executed through the status change of an object that has been defined as PARENT object (through a PASSIVE EVENT) of the RUN-SCRIPT, the used INPUT value depends on the configuration of the RUN-SCRIPT; the choice is between:
    - either the value of the PARENT object itself (that has triggered the execution)
    - or a constant value, which will be sent whenever the related PARENT object changes state

In both cases, the INPUT value can be obtained inside the script by calling the following function:

```
input();
```

Example:

```
$val = input();
```

Depending on the PARENT object, which triggered the execution of the RUN-SCRIPT, the INPUT value can be numeric or a string; the PHP language does not make a big difference of it, since the variable, to which the INPUT value is assigned, will be adapted automatically. The author of the script has to check the type of the input value; for example, in order to check if the input value is a string, the following PHP function can be used:

```
is_string(input())
```

To check if the value is an integer or a floating point number (float), the following PHP functions are recommended:

```
is_int(input())
is_float(input())
```

And at last, to check if the value is a string with numeric characters, you can use:

```
is_numeric(input())
```

Further details on the data types supported in PHP as well as the different control and conversion tools can be found in the following documentation on the web:

```
http://www.php.net/manual/en/ref.var.php
```

Once completed the script will return an OUTPUT value to the run-script as result which can for example be displayed within the visualisation or be used to execute other events. If inside the RUN-SCRIPT at least one

OUTGOING CONNECTION has been defined, the result value will be passed to the CHILD objects through this event.

In order to return a value at the end of the execution, the following function is used:

```
output()
```

Example:

```
output(1);
```

Hint: Calling the function output() will terminate the execution of the script; code in the script that is placed after this function will be ignored.

If at the end of the script no value should be returned, it is enough to call the function without parameters:

```
output();
```

In this case:

- no values is passed to the *RUN-SCRIPT*
- the *OUTGOING CONNECTIONS* are not executed

Hint: The fact that when calling the function output() without parameters no *OUTGOING CONNECTIONS* are executed must not be seen as a restriction, but as a higher flexibility. In this way it is possible to handle the events already within the script and pass different values to the single events, which is not possible when using the returned OUTPUT value (same for all connections). Further details regarding this functionality can be found in the chapter about the "surrounding" library.

2.5 DEBUG, INFO AND ERRORS

It is possible to insert different kinds of output messages into the script: *DEBUG* (to control the correct function of the script during its creation), *INFO* (to display information during the execution of the script) or *ERROR* (to notify when there are errors that prevent a correct termination of the script).

In order to include such messages, the following commands can be used:

```
debug()
info()
error()
```

Example:

```
debug("This is a debug message");
info("This is a info message");
error("This is an error message");
```

The function error() provides also a second optional parameter for the so called RETURN code of the script. This code (not to be confused with the OUTPUT value of the script) is a numeric value, which informs the execution environment if the executed script has been terminated correctly. Normally this value will be 0 (zero), which corresponds to a correct execution without errors. If a script is interrupted by calling the function error(), this code can be used to provide information about the occurred error:

```
error("This is an error message", 1);
```

Your KNXCONTROL device uses return codes which can be used through static constants. The most important are:

RETURN CODE	NUMERIC VALUE	DESCRIPTION
_DPAD_RESULT_NOERROR	0	No error
_DPAD_RESULT_DBERROR	1	Database error
_DPAD_RESULT_SYSERROR	4	General system error
_DPAD_RESULT_CONFIGER-ROR	5	Configuration error
_DPAD_RESULT_USERERROR	6	User error (e.g.: input of unsupported values)
_DPAD_RESULT_IOERROR	7	File system error (flash memory of the product, e.g. error when saving or accessing files)
_DPAD_RESULT_TIMEOUT	12	Timeout error (script execution takes too long)

The constants can be integrated in the error message as shown below:

```
Error ("This is an error message",_DPAD_RESULT_USERERROR);
```

The messages can be very helpful when testing the script (see the chapter "Debug"). On runtime the messages are not shown at all; nevertheless they are stored in the internal log and can be viewed in a second moment.

2.6 INCLUSION OF SCRIPTS

It is also possible to include other scripts inside a customized script; this permits to use previously created scripts like a „private library".

In order to include a script, please use the following command:

```
include_script(<Script Name>);
```

<Script Name> is a placeholder for the script to be included, e.g.:

```
include_script("MyScript");
```

It is also possible to pass an INPUT value to the included script, which will be used instead of the objects connected in the input list. To do so, just pass the value as second argument of the include function:

```
include_script("MyScript","New INPUT value");
```

This function also returns the OUTPUT value of the included script. This value can be stored inside a variable and then be used in the main script:

```
$value = include_script("MyScript");
```

As an example, please check out the script "Double", which simply doubles the INPUT value and returns the result:

```
$double = input() * 2;
output($double);
```

SCRIPT „Double"

If this script now should be used as "library" within another script (for example to double other values), it can be realized as shown below:

```
$double1 = include_script("Double");    //Doubles the INPUT value
$double2 = include_script("Double ",2); //Doubles the value 2 (returns 4)
$double3 = include_script("Double ",5); //Doubles the value 5 (returns 10)
[...]
```

Hint: The INPUT and OUTPUT values of a script that has been included via include_script() do NOT change the INPUT and OUTPUT values of the main script.

In order to include a sample script instead of an own script, please use the following function (same as include_script()):

```
include_sample("SampleScript");
```

3 Scripts

3.1 INTRODUCTION

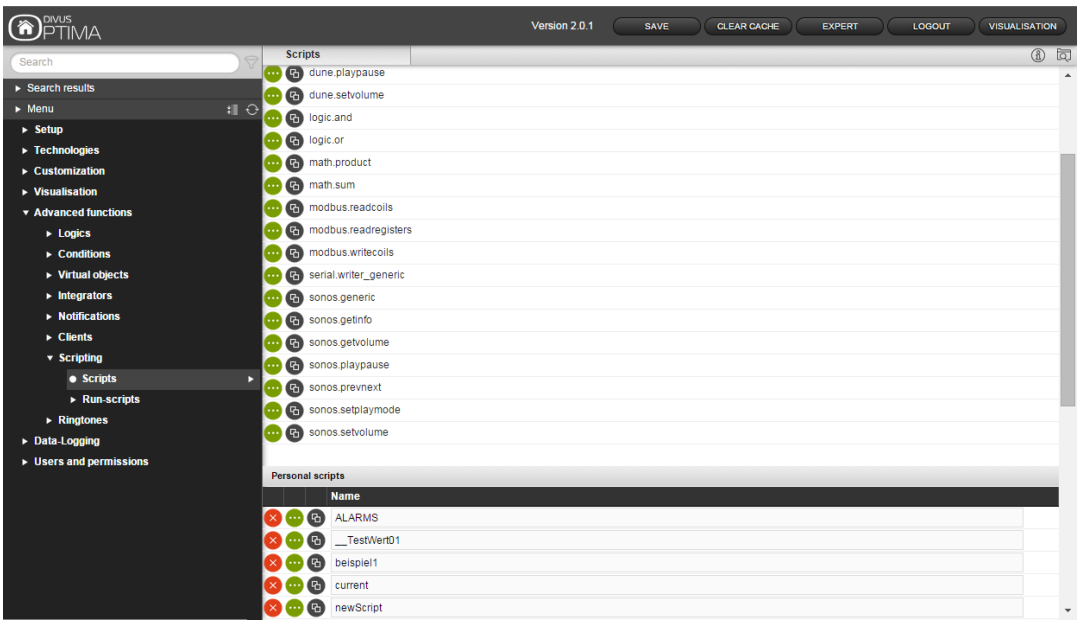
This chapter explains how to create, edit, delete and export customized scripts.

3.2 OVERVIEW PAGE OF THE SCRIPTS

In order to access the overview page of the scripts, please follow the steps below:

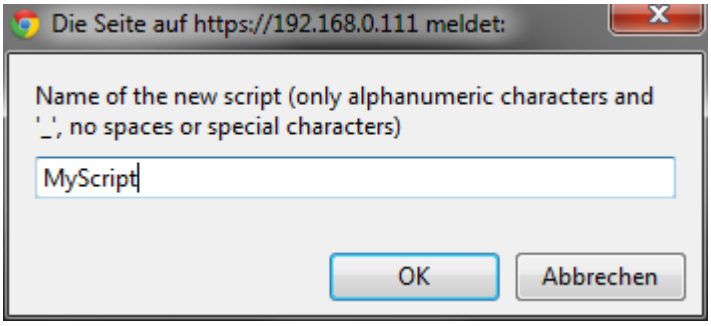
- Log into the OPTIMA user interface with an administrator account
- Open the ADMINISTRATION area
- In the navigation menu, select "ADVANCED FUNCTIONS" → "SCRIPTING" → "SCRIPTS"

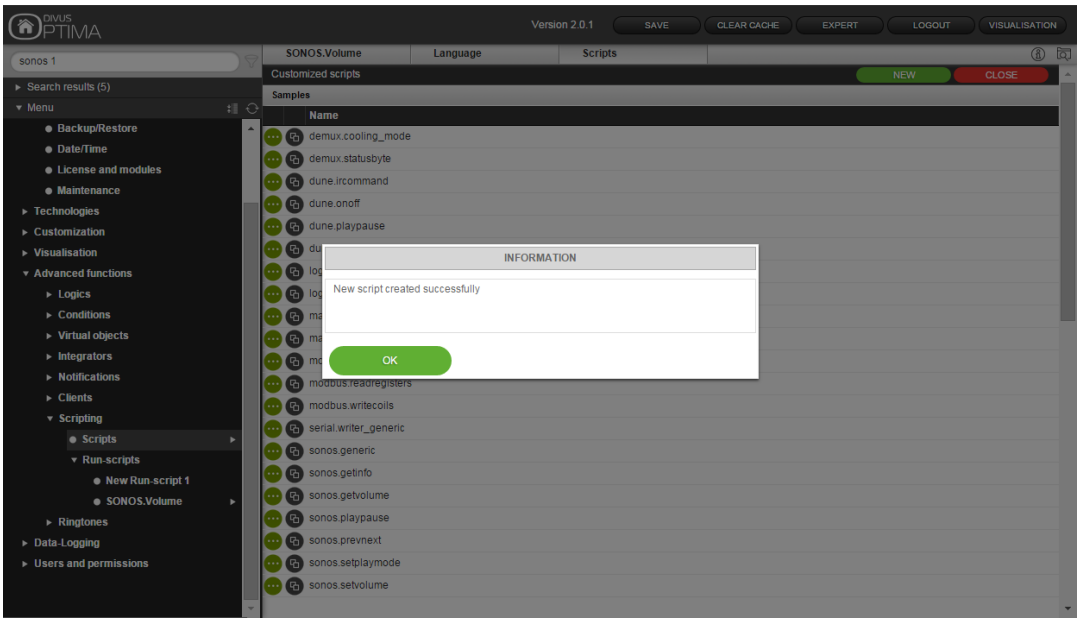
A page similar to the following will be shown:



The upper area of the page shows the sample scripts that are preinstalled on your KNXCONTROL device. Those can't be edited or deleted, but they can be cloned (for example to create your own scripts starting from the samples).

The lower area instead shows the personally created scripts. In order to add a new script, just click on the NEW button at the bottom of the page. After inserting a name for the new script it will be created, which is also confirmed through an on screen message:





If you want to create a new script starting from one of the script examples, just hit the corresponding COPY/CLONE icon (grey) next to the name of the sample script. Define also a name for the copied script.

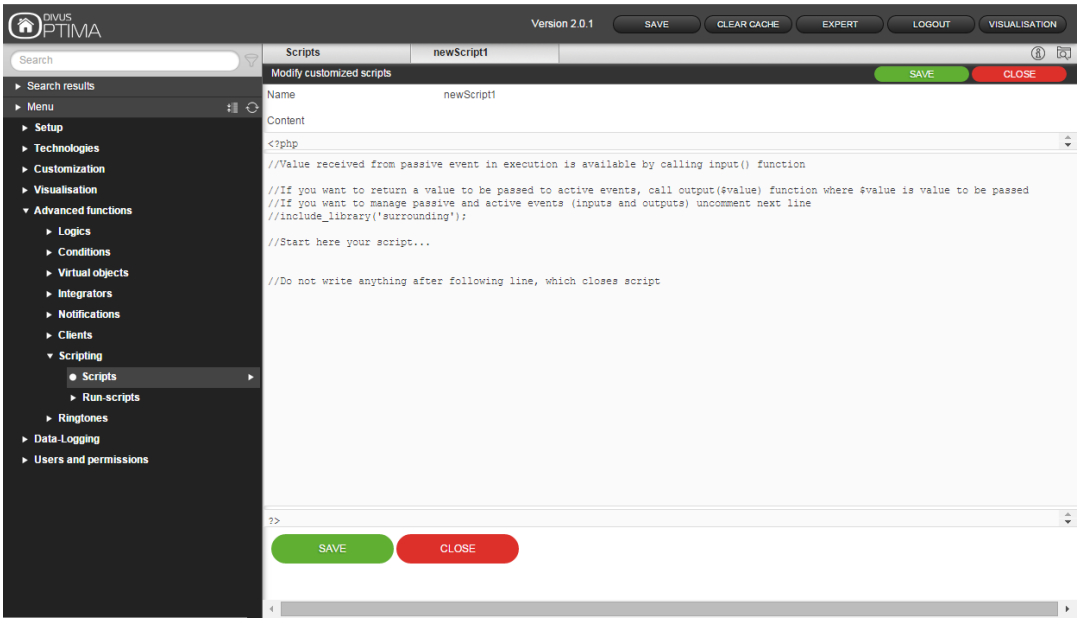
In order to delete a script, just click on the corresponding DELETE button (red). After confirmation of the action the script will be removed from the flash memory of your KNXCONTROL device (this action can't be undone).

If you want to access the detail page of a script, please click on the corresponding EDIT button (green) next to the script's name.

3.3 SHOWING / EDITING SCRIPTS

If the *EDIT* button of a script has been pressed, the detail page of the script will be opened and the PHP code will be shown. When accessing a sample script, this code can't be changed; when accessing a new/cloned script instead, the contents are editable.

The following screenshot shows the detail page of a new script:



As you may recognize, the TAGS for opening the script...

```
<?php
```

... and for closing it...

```
?>
```

... are already integrated (outside of the editable area) and must not be included! The following chapters will give you all the information on how to create scripts using the available libraries.

When finished with editing the code, the changes must be stored through the *SAVE* button. If you exit hitting directly the *CLOSE* button, or switch to another tab inside Optima, all unsaved changes will be lost!

3.4 EXAMPLE

The following code will be used as code example in the further chapters of the manual (if not mentioned differently); it simply returns the received INPUT value after showing it within a debug message (the debug message will be visible only during debug, as also explained in the following chapters):

```
$value = input();
debug("Obtained INPUT value: " . $value);
output($value);
```

4 Run – Scripts

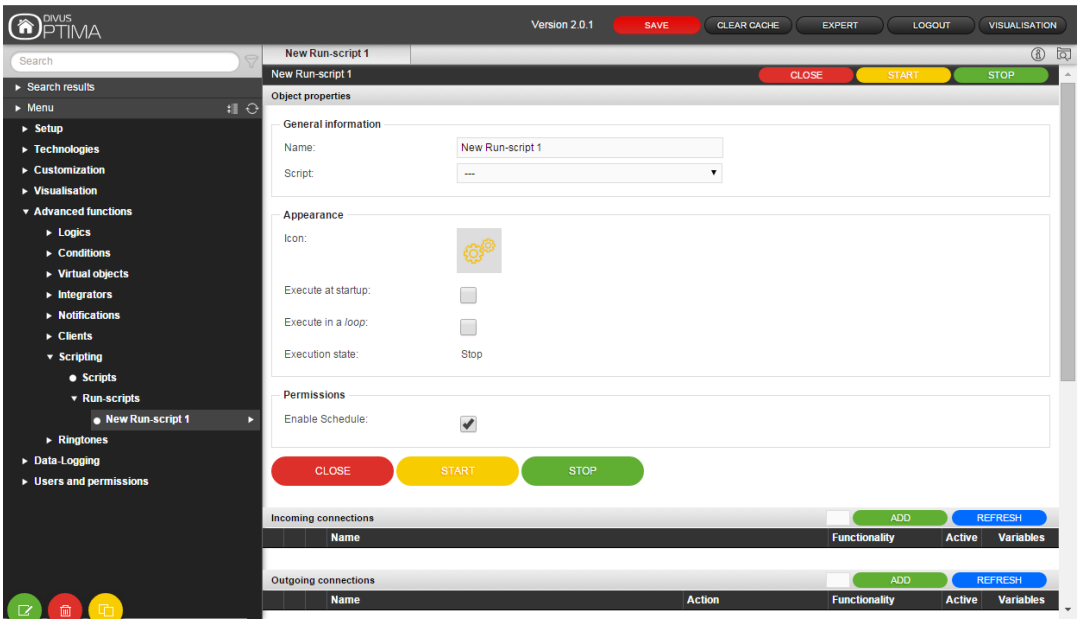
4.1 INTRODUCTION

This chapter shows how to create and manage objects of type RUN-SCRIPT; these objects are used in order to execute the created scripts. In other words, the run-script, wrapped around a PHP script, makes the script an object inside Optima. Then this object can be configured, shown in a room, executed, etc.

4.2 CREATION OF A RUN-SCRIPT

In order to create a new RUN-SCRIPT, please follow the instructions below:

- Log into the OPTIMA user interface with an administration account and open the ADMINISTRATION area
- In the navigation menu, select "ADVANCED FUNCTIONS" → "SCRIPTING" → "RUN-SCRIPTS"
- Press the ADD button in the bottom left toolbar
- Access the configuration window of the RUN-SCRIPT through the EDIT function



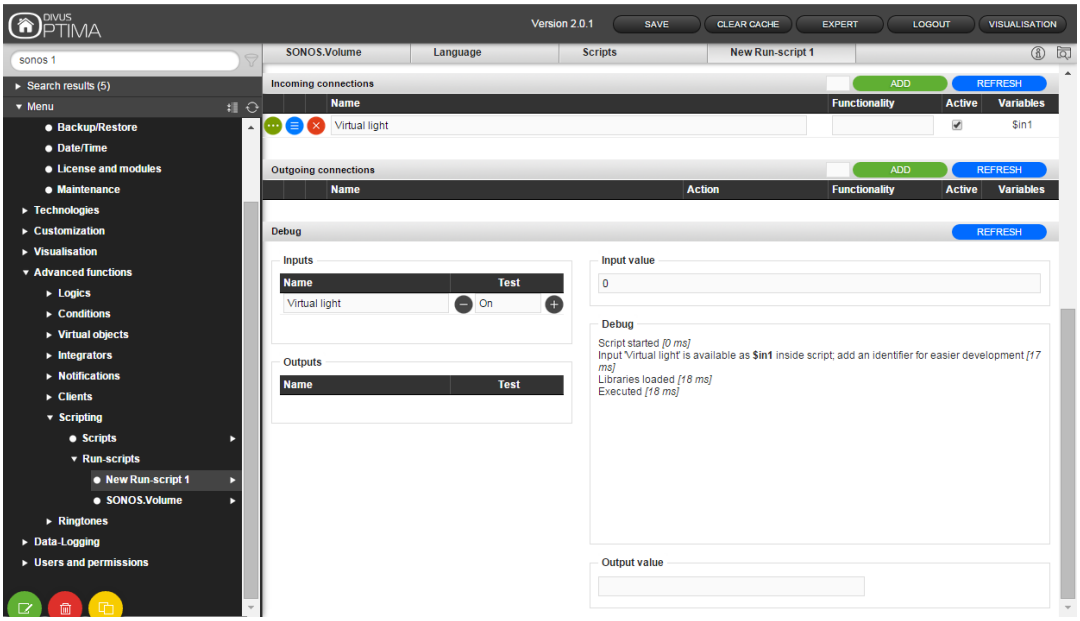
Besides the name of the new run-script, also the script that should be executed must be defined by selecting it from the drop-down menu. Furthermore the graphical design of the run-script can be selected, which defines the representation of the run-script within the pages of the visualisation and also defines the interaction possibilities given to the user.

Furthermore, the following options can be defined for the run-script:

EXECUTE AT STARTUP	When enabling this checkbox, the run-script is executed immediately after the startup of the KNXCONTROL device
EXECUTE IN A LOOP	When enabling this checkbox, the run-script will be executed repeatedly inside a loop, until it is either stopped manually or through an event. Details about this functionality can be found in the chapter „background execution"
LOOP TIME	If the run-script is executed in a loop, please specify here the duration of the loop transition

it is furthermore possible to start and stop a run-script manually, using the corresponding buttons within the detail page.

The lower part of the detail page permits to manage the relations of the run-script with other objects, as well as the testing (DEBUG) of the script in real time.



4.2.1 INCOMING CONNECTIONS

The section *Incoming Connections* allows to define one or more objects that can trigger the execution of the *run-script* when changing their state.

Once an object has been dragged into this area, it must be defined which value should be passed to the script whenever the source object changes its state. Compared to other objects of the software, for which this value can be selected from a drop-down menu, the value in this case must be inserted manually, since only in this case

the maximum flexibility can be granted in order to be able to create concatenations of different run-scripts. This field can contain the following values:

$VAL	The current value of the source object will be passed to the script as input value
$NVAL	The inverted value of the source object will be passed to the run-script as input value
ANY NUMER / STRING	The inserted, constant value will be passed to the run-script as input value, independently from the object's value

In any of these cases the INPUT value can be obtained inside of the script using the following command, as already mentioned in the last chapters:

```
input()
```

Hint: The value within this field is not the only possibility to access the values of connected objects from within the script. The „surrounding" library offers the possibility (see corresponding chapter of this manual) to access the state of every single object connected as input of the script. Therefore, the *VALUE* column is only important for the input value of the script which can be accessed through the command input(), as already seen.

Each object can have a so called identifier assigned (consisting of a text string without spaces or special characters – column *Functionality*), which simplifies the usage of the object within a script with active *SURROUNDING* library. As described more in detail in the next chapter, the following command permits to call the input object within the script by using this identifier:

```
$me->getParentByIdentifier("identifier")
```

The column *variable* shows the name of the variable, under which the INPUT object can be reached within the script.

Hint: The column *variable* is not always refreshed automatically. Use the "REFRESH" button if it should not show correct values.

4.2.2 OUTGOING CONNECTIONS

In the same way as the incoming connections, also the OUTGOING CONNECTIONS can be defined through the corresponding section. Objects inserted into this section can be commanded in dependency of the result of the executed run-script. For each object it is possible to define the action to be executed, as well as the value that should be set (if supported). The value can also be set to the "*VALUE OF THE RUN-SCRIPT*"; in that case, the *OUTPUT* value of the script will be passed to the object(s). The output is passed by the following command:

```
output(…);
```

Hint: As already mentioned, the execution of actions on the objects defined as outputs can be inhibited by not

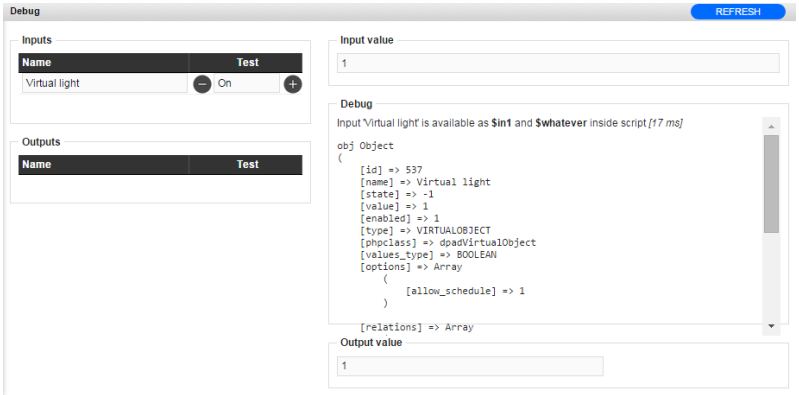
passing any argument to the function output(). In this case, the script itself must handle the execution of the desired actions, as described more in detail in the chapter about the "surrounding" library.

4.3 DEBUG

Once the desired objects have been defined as inputs/outputs (i.e. incoming/outgoing connections) of the run-script, the execution of the run-script can be tested thanks to the *DEBUG* function; this function executes the script (all actions within the script are executed at 100%), but ignores the execution of actions on the connected output objects (*CHILD*-objects), whose state will be shown within the column TEST, but actually NOT really set.

In order to simulate status changes of objects in the *Incoming connections* section, you can simply change the value within the column *TEST*. The selected value will be automatically taken over from the field *Input value* and passed to the script as *INPUT* value.

If there are any messages (debug, info, ...) present in the script, they will be shown inside the DEBUG window. This window generally shows all message outputs of the script, including messages realized using other PHP functions like for example print(), echo(), print_r() etc.:



The value returned by the output() function of the script will be shown in the field *Output value*; furthermore the column *TEST* of the objects in the output will be changed according to the configuration of the run-script.

As alternative it is also possible to pass values to the script manually. In this case, just write the values into the *INPUT VALUE* field and press the "TAB" key on your keyboard (or the *ENTER* key or, even easier, just click on a free space outside of the input field).

Hint: If the *INPUT VALUE* doesn't correspond to the value shown in the *TEST* column of the object connected as input or, vice-versa, the value in the *TEST* column of an object connected as output doesn't correspond to the value shown in *Output value*, please check your script as well as your inputs in the *VALUE* column of the different objects.

The simulation function not only is useful to debug the run-script and its various relations / connections, but also to debug the PHP code of the script file itself. For this purpose, the generous use of *DEBUG* messages can be really helpful and is recommended.

4.4 BACKGROUND EXECUTION

Run-scripts can also be executed in background, for example in order to repeat certain actions (e.g. periodically requesting a value), without depending on user interactions.

The first of these options in the detail page of the run-script permits to enable the automatic start of the run-script at completed startup of the software. If enabled, the script will be started whenever the KNXCONTROL device is started or restarted.

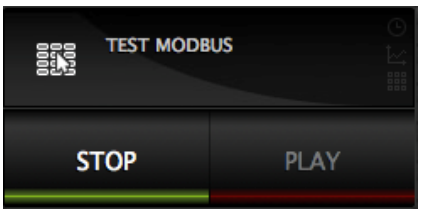
If the option "EXECUTE IN A LOOP" is enabled, the script will be automatically repeated, as long as it is either stopped manually or through an EVENT. If this option is combined with the first option, it is possible to realize a continuous execution of the script from the start of the visualisation, which runs completely hidden and requires no interaction from the user.

If the loop execution is enabled, it is also necessary to define a loop timer, which determines the duration of each iteration of the loop. The scripting routine automatically calculates the execution time of the script and subtracts this value from the inserted loop time; in this way, each loop cycle will take exactly as long as the inserted time value (with a minimum tolerance).

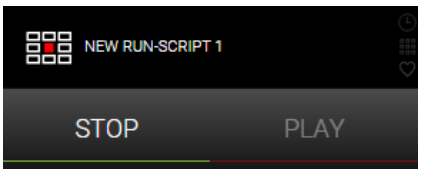
Hint: When enabling the loop execution of a script, the OUTPUT VALUE of the script (generated by the function output()) is automatically used as INPUT VALUE for the next iteration. In this way, the script can pass over the values incrementally; if this is not desired, please DON'T call the functions input() and output() inside the script!

4.5 REPRESENTATION IN THE VISUALISATION

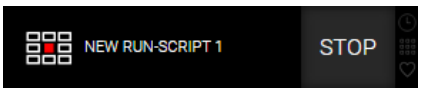
If a run-script is added to a page of the visualisation, its representation corresponds to the one shown in the screenshot(s):



(Optima 1.3.x)

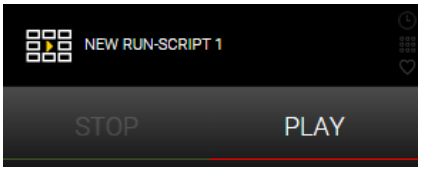


(Optima 2.x – Expanded)



(Optima 2.x – Compact)

When clicking on the *PLAY* button, the run-script is executed and the shown symbol will change for the duration of the execution. If the run-script is executed in a loop, the object will remain in execution as long as the *STOP* button is pressed or the system interrupts the script execution.



Hint: The interruption of a run-script that is executed in a loop will occur only after the execution of the script code, which means at the end of the current iteration. Therefore, it can happen that the run-script seems to remain in execution although the STOP button was pressed. This depends on the time the system needs to run through the code of the script.

5 „Object“ Library

5.1 INTRODUCTION

The “object” library permits to manage one or more objects of the visualisation of the KNXCONTROL device within a script, in form of variables. Those variables are at 100% PHP objects and therefore offer different attributes and methods, which can be used to interact with the rest of the software. For example, it is possible to read out the values of each object in the software in real time or to execute actions (like e.g. commanding KNX functions, executing scenarios, etc.).

5.2 INCLUSION OF THE LIBRARY

In order to include the library, use the following command at the start of the script:

```
include_library(“object”);
```

Starting from the next line, the following class can be used within the script:

```
objM
```

This class (“object manager”) offers different functions (or “methods”), which permit to load objects from the software and execute operations on them. The class objM is used through “static calls”, like for example:

```
objM::objGet(...);
```

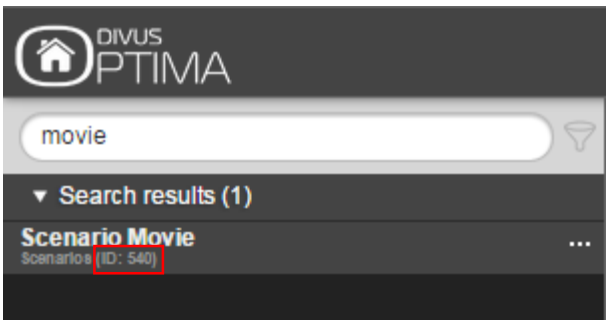
As visible, the static calls start with the name of the class (without the typical variable prefix “$”), followed by two colons and the name of the function to be called; these calls are different from normal object calls (whose names always have the prefix “$”), which use the separator “->” in front of function calls.

5.3 LOADING AN OBJECT

In order to load an object as a variable into a script, the following function can be used:

```
$variableName = objM::objGet(XXX);
```

XXX is the unique numeric ID that identifies the object within the software. The ID of an object can be found using the search function within the configuration area of the software, as shown in the following screenshot:



For the object "Scenario Movie" shown in the screenshot, the function would look like below:

```
$scenarioMovie= objM::objGet(540);
```

The variable now contains a reference to the selected object and offers the following attributes within the script:

ATTRIBUTE	DESCRIPTION	EXAMPLE/ACCESS
id	ID of the object	$obj->id
name	Name of the object within the visualisation	$obj->name
state	Current state of the object. NOTE: this attribute is only usable for scenario objects; in this case: 1 → Scenario stopped 2 → Scenario in execution	$obj->state
value	Current value of the object NOTE: this attribute can be used for all object types of the software that have a value, except scenarios (see attribute „state")	$obj->value
enabled	Informs whether the object is active or not. Possible values: 0 , 1	$obj->enabled
msp	Most important parameter of the object, depends on its type. Some examples: KNX objects: group address Logics: logical expression	`$obj->msp`

	Scenarios: number of iterations	
type	Function object type of the object, is used by the communication service. Some examples: KNX objects: "EIBOBJECT" Virtual objects: "VIRTUALOBJECT" Scenarios: "SCENARIO" Logics: "LOGIC" Conditions: "CONDITION" Run-scripts: "CSCMD" *NOTE: a complete list of the available types can be found in the appendix, chapter 12.1*	`$obj->type`
phpclass	Graphical web object type, is used by the web interface. Some examples: KNX objects: "dpadEibObject" Virtual objects: "dpadVirtualObject" Scenarios: "dpadScenario" Logics: "dpadLogic" Conditions: "dpadCondition" Run-scripts: "dpadScriptRunner" *NOTE: the connection between a "type" and a "phpclass" is not always unique: in some cases multiple "phpclass" refer to the same "type". A complete list of the available types can be found in the appendix, chapter 12.1*	`$obj->phpclass`
values_type	Data type or encoding. *NOTE: depends on the object type*	`$obj->values_type`

options	Array with optional values. *NOTE: depends on the object type*	`$obj->options` `$obj->options["..."]`

5.4 RELATIONS WITH OTHER OBJECTS

In order to recognize relations between an object and other objects, either of the type "PARENT" ("PASSIVE EVENTS" for the reference object) or of the type "CHILD" ("ACTIVE EVENTS" for the reference object), the following methods are available, which must be called directly on the variable with the object reference (and not on the class "objM"):

```
$object->loadParents();
$object->loadChilds();
```

Both functions return arrays with the corresponding values:

```
$object->parentRelations[];
$object->childRelations[];
```

Those arrays contain a PHP object of type "relation" for reach relation to other objects (*parent* or *child* relations), which offers the following attributes:

ATTRIBUTE	DESCRIPTION	EXAMPLE/ACCESS
id	ID of the relation within the database NOTE: this ID is the one of the relation itself and must not be confused with the object ID of the connected objects („PARENT" or "CHILD", see below)	`$relation->id`
parent child	Reference to the objects that are connected either as "PARENT" or as "CHILD" NOTE: these objects are complete PHP objects with the same attributes as described in chapter 5.3	`$relation->parent` `$relation->child`
condition	In case of EVENT relations (ACTIVE or PASSIVE), this is the filter on the PARENT object that – if fulfilled – toggles the execution of the event.	`$relation->condition`

	On “static” or “structural” relations (e.g. connection of an object to a room) this field is not used and normally contains the expression “NO-CONDITION”	
action	In case of EVENT relations (ACTIVE or PASSIVE), this is the ACTION to be executed on the CHILD object. On “static” or “structural” relations (e.g. connection of an object to a room) this field is not used and normally contains the expression “NO-ACTION”	`$relation->action`
value	In case of EVENT relations (ACTIVE or PASSIVE), this is the VALUE to be passed to the CHILD object. On “static” or “structural” relations (e.g. connection of an object to a room) this field is not used and normally contains the expression “NO-VALUE”	`$relation->value`
enabled	Informs whether the object is active or not. Possible values: 0 , 1	`$relation->enabled`
options	List of optional settings of the relation; these settings, if present, are shown in the format ... param1='value1'|param2='value2'|... ... , as concatenation using the separator “|”. Numeric values can also appear without apostrophe.	`$relation->options`

Hint: The attributes described here correspond to the values of the database table *DPADD_OBJECT_RELATION*, which is used from the software to handle relations between objects.

Hint: The functions *loadChilds()* and *loadParents()* normally only load relations of the type EVENT, which are most commonly used within scripts. In order to load all kind of relations, please just pass the argument true to the function:
$object->loadChilds(true);
$object->loadParents(true);

If only a certain type of relations should be returned, just past false as first argument and an array of the desired relation types as second argument:

$object->*loadChilds(false,[“GENERIC_RELATION”,“ACTION_RELATION”]);*

$object->loadParents(false,["GENERIC_RELATION","ACTION_RELATION"]);

Please note that the attributes "parent" and "child" of a relation also are full PHP objects, with the same set of attributes as the starting objects. For example, in order to access the value of the first CHILD object of the starting object, the following expression can be used:

```
$firstChildValue = $object->childRelations[0]->child->value;
```

where:

$object->childRelations[0]	First CHILD relation (Index 0)
$object->childRelations[0]->child	Reference to the CHILD object of the relation
$object->childRelations[0]->child->value	Value of the CHILD object

In order to understand how an object and its relations to PARENT and CHILD objects are structured, the following expression can be helpful (always assuming that $object is the starting object that should be analyzed):

```
echo "<pre>"; print_r($object); echo "</pre>";
```

The command print_r (surrounded by the HTML tags "<pre>" and "</pre>", which format the output and make it readable) during the execution of the script (using a run-script, as explained it chapter 4.3) will use the DEBUG window in order to show the complete structure of the object given as argument.

Here an example of such an output for an object of type "Scenario":

```
obj Object
(
   [id] =>540
   [name] =>Scenario Movie
   [state] => 1
   [value] =>
   [enabled] => 1
   [options] => Array
      (
      )

   [relations] => Array
      (
      )

   [childRelations] => Array
      (
         [0] => relation Object
            (
               [id] => 872
               [parent] =>obj Object
 *RECURSION*
               [child] =>obj Object
                  (
                     [id] => 487
                     [name] =>Light - Living room
                     [state] => -1
```

```
                        [value] => 0
                        [enabled] => 1
                        [options] => Array
                           (
                              [category] =>lighting
                           )

                        [relations] => Array
                           (
                           )

                        [childRelations] => Array
                           (
                           )

                        [childsLoaded] =>
                        [parentRelations] => Array
                           (
                           )

                        [parentsLoaded] =>
                        [msp] => 0/0/4
                     )

                  [condition] => NO-CONDITION
                  [action] => NO-ACTION
                  [value] => 0
                  [enabled] => 1
                  [options] => Array
                     (
                     )

               )

            [1] => relation Object
               (
                  [id] => 873
                  [parent] =>obj Object
*RECURSION*
                  [child] =>obj Object
                     (
                        [id] => 491
                        [name] =>Light - Kitchen
                        [state] => -1
                        [value] => 1
                        [enabled] => 1
                        [options] => Array
                           (
                              [category] => lighting
                           )

                        [relations] => Array
                           (
                           )

                        [childRelations] => Array
                           (
                           )
```

```
                        [childsLoaded] =>
                        [parentRelations] => Array
                            (
                            )

                        [parentsLoaded] =>
                        [msp] => 0/0/6
                    )

                [condition] => NO-CONDITION
                [action] => NO-ACTION
                [value] => 0
                [enabled] => 1
                [options] => Array
                    (
                    )

            )

         [2] => relation Object
            (
                [id] => 875
                [parent] =>obj Object
*RECURSION*
                [child] =>obj Object
                    (
                        [id] => 503
                        [name] =>Light Dimmer
                        [state] => -1
                        [value] =>  75
                        [enabled] => 1
                        [options] => Array
                            (
                                [category] => lighting
                            )

                        [relations] => Array
                            (
                            )

                        [childRelations] => Array
                            (
                            )

                        [childsLoaded] =>
                        [parentRelations] => Array
                            (
                            )

                        [parentsLoaded] =>
                        [msp] => 0/0/15
                    )

                [condition] => NO-CONDITION
                [action] => NO-ACTION
                [value] => 0
                [enabled] => 1
```

```
                [options] => Array
                    (
                    )

            )

        )


    [childsLoaded] => 1
[parentRelations] => Array
        (
        )

    [parentsLoaded] => 1
)
```

As you can see, the scenario contains 3 relations with *CHILD* objects (present in the Array childRelations, with index 0,1 and 2); each of these relations, beneath the different attributes also contains a reference to the CHILD object, which contains a set of attributes of its own.

Please note that the relations also show the reference to the PARENT object; since in this case the PARENT object is the starting object, the reference is automatically renamed by the function print_r and displayed using the following expression:

```
*RECURSION*
```

Hint: The command **print_r**, even if very useful during the creation and debugging of the script (in order to understand the object structure or, more generally, the data structure), during the „real" usage of the script is completely useless (like every other message output function) and just slows down the execution of the script. Therefore it is recommended to disable every kind of message output function once the script is working correctly.

It is furthermore possible to request an array of the objects by using the property of the relation to another object ($object the downstanding example). To do so, use one of the following commands:

```
$object->getParentsByRelationField($fieldName,$fieldValue);
$object->getChildsByRelationField($fieldName,$fieldValue);
```

where:

- $fieldName Name of the property of the relation that should be used as filter
- $fieldValue Value of the property

If you want to get a reference to the relation itself (instead of the objects), you can use the following commands:

```
$object->getParentRelationsByField($fieldName,$fieldValue);
$object->getChildRelationsByField($fieldName,$fieldValue);
```

The arguments that need to be passed are the same as above.

Furthermore it is also possible to get a reference to the object (connected either in the Incoming connections or in the Outgoing connections) by using the manually assigned identifiers (check out chapter 4.2.1) and the following commands:

```
$object->getParentByIdentifier($identifier);
$object->getChildByIdentifier($identifier);
```

5.5 COMMANDING AN OBJECT

The object manager objM permits to send commands to any kind of object in the same way as is can be done through the visualisation pages of the OPTIMA interface or through logics, scenarios etc.

The following function shows the general method for the execution of operations on objects:

```
objM::objPerformOperation(ID,OPERATION,VALUE);
```

where

- ID — ID of the object, on which the operation should take place
- OPERATION — Operation to be executed
- VALUE — If supported, the value that is passed to the object

Example: in order to turn on a KNX object of type ON/OFF (the object in this example will use the ID "123"), the following function must be called...

```
objM::objPerformOperation(123,"SETVALUE",1);
```

... while for the execution of the scenario object seen before (ID 540) the following expression is enough...

```
objM::objPerformOperation(540,"EXECUTE");
```

... without having to pass a value as argument, since the execution of scenarios doesn't require any values, as explained on the next page.

The following table shows the operations (and the values, if supported) for the most important objects of the software:

OBJECT	OPERATION	DESCRIPTION	VALUE
KNX object	SETVALUE	Sends a definable value to the KNX group address	Depends on the KNX object Examples:

			0 → OFF 1 → ON 50 → set to 50%
	GETVALUE	Sends a request to get the current state	No value
Scenario	EXECUTE	Executes the scenario	No value
	STOPEXECUTION	Interrupts the scenario (if in execution)	No value
Logic	EVALUATE	Evaluates the logic and sets the outputs again	No value
Condition	CALCULATE	Evaluates the condition and sets the outputs again	No value
On-Screen Notification	INSERT	Puts the notification into the log table, in order to show it inside the message central	No value
E-Mail notification	SENDMAIL	Sends out the mail notification	No value
Integrator	INTEGRATE	Calculates the value of the integrator again	No value
	RESET	Resets the integrator value to 0	No value
Virtual object	SETVALUE	Sets the value of the virtual object	Depends on the type of virtual object

In order to simplify the work with the mostly used objects (KNX objects, scenarios and virtual objects), the following „shortcuts" have been realized: this functions permit to be executed directly on the PHP variable that contains the object reference:

```
$object->set(...);  //Sets the value passed as argument (SEVALUE operation)
$object->run();     //Executes the scenario
$object->stop();    // Interrupts the scenario
```

If for example we want to start the sample scenario already seen before, the following function calls within the script can be used:

```
$scenarioFilm = objM::objGet(540);
$scenarioFilm->run();
```

5.6 REFRESHING OBJECTS WITHIN THE DATABASE

The following function permits to send out a SQL query, which can change attributes of the object directly within the database table:

```
objM::objUpdateToDb($objectId,$field,$value);
```

The available arguments are listed below:

$OBJECTID	ID of the object to be refreshed
$FIELD	Name of the database column that should be refreshed
$VALUE	New value to be written into the database

In this way it is for example possible to update the name of an object ...

```
objM::objUpdateToDb(XXX,"name","New Name");
```

... where XXX naturally refers to the ID of the object whose name should be changed. The same function can also be called directly on an object. In this case, the correct syntax is:

```
$object->updateToDb($field, $value);
```

It is furthermore possible to read the attributes of an object from the database during the execution of the script and to force their refresh:

```
$object->refreshFromDb($field);
```

Example: in order to refresh the value of the object, you can use the following command:

```
$object->refreshFromDb("value");
```

Note: *value* here is the name of a database field, not a generic value, and tells Optima to load its content and refresh the text shown in the visualisation. So, if the content changed, the visualisation will reflect that change.

6 „Surrounding“ - Library

6.1 INTRODUCTION

The library “surrounding” provides objects and functions that can be used to interact directly with the run-script and the objects defined as Incoming connections and Outgoing connections.

This library depends directly on the previously seen library “object”; the structure of the objects and relations within this library therefore rely on the explanations in chapter 5, to which will be referred for some details.

Hint: If the “surrounding” library is included in a script, the “object” library must NOT be included separately, since it wil already be included automatically.

6.2 INCLUSION OF THE LIBRARY

In order to include the library, use the following command at the start of the script:

```
include_library(“surrounding”);
```

6.3 ENVIRONMENT OF THE SCRIPT

Starting from the next line, the following object can be used within the script:

```
$me
```

This object is structured in the same way as the objects described in chapter 5 and is a direct reference to the run-script that executes the script in question (and therefore allows a direct interaction). The object naturally permits to access the objects connected as Incoming connections (“PARENT objects”) and Outgoing connections (“CHILD objects”) in the configuration page of the run-script and gives them back as an array (please check out chapter 4.2).

```
$me->parentRelations[];   //INPUT relations
$me->childRelations[];    //OUTPUT relations
```

The inputs of the run-script (“PARENT objects”) can also be accessed directly using the following variables:

```
$in1;     //First input of the run-script
$in2;     //Second input of the run-script
```

```
…
```

In the same way also the OUTPUTS of the run-script ("CHILD objects") can be accessed directly:

```
$out1;      //First output of the run-script
$out2;      //Second output of the run-script
…
```

Hint: As already recommended in chapter 5.4, while debugging the script, the whole structure of the run-script can be displayed using the following expression:
echo "</pre>"; print_r($me); echo "<pre>";

6.4 EXAMPLES

If a script should be created that as OUTPUT VALUE shows the sum of the values of two input objects, it can be realized in the following way (assuming that the corresponding objects have been defined as Incoming connections of the run-script):

```
include_library("surrounding");       //Library inclusion
$sum = $in1->value + $in2->value;     //Sum of the values of the first two inputs
output($sum);                        //Returning the calculated value
```

If now also an object connected to the OUTPUT of the run-script should be set to the calculated value directly from within the script (instead of using the returned OUTPUT value and an Outgoing connection), the script may be changed in the following way:

```
include_library("surrounding");      //Library inclusion
$sum = $in1->value + $in2->value;    //Sum of the values of the first two inputs
$out1->set($sum);                   //Setting the value of the first output
output();                          //Script termination with no return
```

In the second case it is very important to call the function output() WITHOUT an argument, in order to prevent the run-script from executing the connected events (since the desired action was already done inside the script). Otherwise the output would be commanded twice: the first time directly from within the script and the second time through the execution of the event put into the Outgoing connections of the run-script.

A script that calculates the sum of all the Incoming connections of the run-script might look like the following:

```
include_library("surrounding");                //Inclusion library
$sum = 0;                                    //Init sum with value 0
foreach($me->parentRelations as $parentRelation)   //Check of the input relations
{
    $parent = $parentRelation->parent;          //Identification of the input
    $sum = $sum + $parent->value;              //Adding the value to the previous
}
output($sum);                                //Returning the calculated value
```

Hint: In the very same way the sample scripts AND and OR preinstalled in your KNXCONTROL device have been realized; further information on these scripts can be found in the chapter about the sample scripts.

7 „Serial“ Library

7.1 INTRODUCTION

The library “serial” permits to read and write data over the RS232 interface of your KNXCONTROL device or, in combination with a compatible USB-RS232 adapter, also over one of the available USB ports.

7.2 INCLUSION OF THE LIBRARY

In order to include the library, use the following command at the start of the script:

```
include_library(“serial”);
```

The library offers the following class...

```
$serialM
```

... which permits to handle operations on the RS232 interface , as described in this chapter.

```
include_library(“serial”);
```

Differently from the classes seen until now (which are called in a static way), the class $serialM must be handled like an object, using the separator “->” instead of “::” before calling functions.

7.3 INITIALIZING THE INTERFACE

The class serialM permits to configure the primary communication parameters of the serial interface; the following table shows the available attributes, the functions needed to access them, the default values and also possible alternative values:

ATTRIBUTE	FUNCTION CALL	DEFAULT	POSSIBLE VALUES
COM port	$serialM->deviceSet(...);	“/dev/ttyS4”	“/dev/ttyS4” (RS232)
		NOTE: RS232 of KNXSERVER	“/dev/dlabusb1” (USB1)

			"/dev/dlabusb2" (USB2)
Baud rate	$serialM->confBaudRate(...);	19200	110, 150, 300, 600, 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200
Parity	$serialM->confParity(...);	"none"	"none", "odd", "even"
Data bits	$serialM->confCharacterLength(...);	8	8, 7
Stop bit	$serialM->confStopBits();	1	1, 1.5, 2
Flow control	$serialM->confFlowControl();	"none"	"none", "rts/cts", "xon/xoff"

If the default values don't need to be changed for your communication type, you can ignore the following functions. Otherwise it is necessary to adjust the attributes before opening the serial interface for read/write operations:

Examples:

```
include_library("serial");
$serialM->confBaudRate(9600);
$serialM->confParity("odd");
$serialM->confFlowControl("rts/cts");
...
```

7.4 WRITING ON SERIAL

In order to start a write operation, the interface must first be opened using the following command

```
$serialM->deviceOpen();
```

Afterwards, using the command...

```
$serialM->sendMessage(...);
```

... it is possible to pass a string as an argument, which will be sent over the serial interface. If necessary, you can also add a termination character at the end of the message:

Examples:

```
$serialM->sendMessage("test");          //Writing "test" with no termination
$serialM->sendMessage("test\n");        //Writing "test" and "new line"
$serialM->sendMessage("test".chr(13)); //Writing "test" and ENTER
```

The function supports also a second argument, through which it is possible to add a time delay (in seconds), which has to pass before read actions can be done on the serial interface (details about reading data from serial can be found in the next chapter):

```
$serialM->sendMessage("test\n",2);    //Write, then wait 2 seconds
$serialM->sendMessage("test\n",0.1);  //Write, then wait 100 milliseconds
```

At concluded write operation, the serial interface must be closed again using the command below:

```
$serialM->deviceClose();
```

7.5 READING FROM SERIAL

Also before starting read operations, the interface must be opened using the following command:

```
$serialM->deviceOpen();
```

Reading data from the serial port can be necessary after write operations with delay (e.g. when awaiting an answer), like already mentioned before. In order to get the data from the serial port, one of the following methods must be used:

```
$value = $serialM->readPort();    //Reads data from serial as long as available
$value = $serialM->readPort(512); //Reads a certain amount of bytes from the serial
```

At concluded read operation, the serial interface must be closed again, as already seen before:

```
$serialM->deviceClose();
```

7.6 DIRECT ACCESS TO THE INTERFACE

In alternative to the functions of the "serial" library, write operations on the serial interface can also be done using the following native PHP expressions:

```
$fp = fopen('/dev/ttyS4','r+b');  //Opens the serial in write mode
```

```
$msg = "...";                    //Initializes the message
fwrite($fp,$msg . Chr(13));        //Writes out the message, followed by ENTER
fclose($fp);                     //Closes the port again
```

If these operations are executed after the initialization of the serial port (see chapter 7.3), also the native PHP commands will use the correct communication settings.

8 „Modbus“ Library

8.1 INTRODUCTION

The “modbus” library permits to manage one or more MODBUS SLAVE devices within a script, using either MODBUS TCP or UDP.

8.2 INCLUSION OF THE LIBRARY

In order to include the library, use the following command at the start of the script:

```
include_library(“modbus”);
```

Starting from the next line, the following class can be used within the script:

```
modbusM
```

This class (“MODBUS manager”) offers different functions (or “methods”), which permit to load objects from the software and execute operations on them. The class modbusM is used through “static calls”, like for example:

```
modbusM::bind(...);
```

As visible, the static calls start with the name of the class (without the typical variable prefix “$”), followed by two colons and the name of the function to be called; these calls are different from normal object calls (whose names always have the prefix “$”), which use the separator “->” in front of function calls.

8.3 ASSIGNING THE „MODBUS SLAVE“ DEVICE

Before any read / write operations can take place, the class modbusM must be connected to the MODBUS SLAVE device used for the communication. This is done using the following command:

```
modbusM::bind($host, $protocol);
```

where:

$host	IP address of the MODBUS device

$protocol	protocol to be used, either "TCP" (default) or "UDP"

As long as this function call is not repeated using different arguments, all operations will be sent to the specified MOD-BUS SLAVE device.

8.4 READING REGISTERS

In order to read one or more registers from a MODBUS SLAVE device, the following command must be used:

```
$data = modbusM::readRegisters($unitID, $reference, $quantity);
```

where

$unitID	ID of the connected device. If not specified explicitly from the manufacturer, use 0 (zero)
$reference	Register to be read. NOTE: the specified value needs to consider the device-internal offset, depending on the configuration. Normally the value "0" would point to the internal register 40001.
$quantity	Quantity of registers to be read. Specify 1 if only one register per call should be read.

The result will be returned by the function in form of a byte array (or, in other words, a string); further information regarding the conversion of the obtained data can be found in chapter 8.9 of this manual.

8.5 READING COILS

In order to read one or more coils (binary in-/outputs) from a MODBUS SLAVE device, the following command must be used:

```
$data = modbusM::readCoils($unitID, $reference, $quantity);
```

Where:

$unitID	ID of the connected device. If not specified explicitly from the manufacturer, use 0 (zero)
$reference	Coil to be read. NOTE: the specified value needs to consider the device-internal offset, depending on the configuration.
$quantity	Quantity of coils to be read. Specify 1 if only one coil per call should be read.

The result will be returned by the function in form of an array with Boolean values (one value for each coil).

8.6 WRITING REGISTERS

In order to write one or more registers to a MODBUS SLAVE device, the following command must be used:

```
$data = modbusM::writeRegisters($unitID, $reference, $data, $dataTypes);
```

where:

$unitid	ID of the connected device. If not specified explicitly from the manufacturer, use 0 (zero)
$reference	Register to be written NOTE: the specified value needs to consider the device-internal offset, depending on the configuration.
$data	Value array that should be passed to the registers (one value for each register to be written).The amount of values defines the amount of registers to be written.
$datatypes	Format array for each register to be written (fitting the elements in $data). Possible values: • "WORD" • "INT" • "DINT" • "REAL"

The function gives back a so called return code: it equals _SCRIPT_RESULT_NOERROR (0) at successful termination or an error code when the script was not finished correctly. Further information on the available return codes can be found in the chapter 2.5 of this manual.

The following code shows an example of the presented function:

```
$data = array(10,-1000,2000,3.0);
$dataTypes = array("WORD","INT","DINT","REAL");
$result = modbusM::writeRegisters(0, 12288, $data, $dataTypes);
```

8.7 WRITING COILS

In order to write one or more coils to a MODBUS SLAVE device, the following command must be used:

```
$data = modbusM::writeCoils($unitID, $reference, $data);
```

where:

$unitid	ID of the connected device. If not specified explicitly from the manufacturer, use 0 (zero)
$reference	Coil to be written NOTE: the specified value needs to consider the device-internal offset, depending on the configuration.
$data	Array with Boolean values that should be passed to the coils. The amount of values defines the amount of coils to be written.

The function gives back a so called return code: it equals _SCRIPT_RESULT_NOERROR (0) at successful termination or an error code when the script was not finished correctly. Further information on the available return codes can be found in the chapter 2.5 of this manual.

The following code shows an example of the presented function:

```
$data = array(TRUE, FALSE, TRUE, TRUE, FALSE, TRUE, TRUE, TRUE,
TRUE, TRUE, TRUE, TRUE, FALSE, FALSE, FALSE, FALSE,
            FALSE, FALSE, FALSE, FALSE, TRUE, TRUE, TRUE, TRUE,
            TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE);
$result = modbusM::writeCoils(0, 12288, $data);
```

8.8 READ AND WRITE REGISTERS

In order to write one or more registers to the MODBUS SLAVE device and contemporarily read out one or more registers (using the same connection), the following command can be used:

```
$data = modbusM::readWriteRegisters($unitID, $referenceRead, $quantity,
$referenceWrite, $data, $dataTypes);
```

where:

$unitID	ID of the connected device. If not specified explicitly from the manufacturer, use 0 (zero)
$referenceRead	Register to be read

$quantity	Amount of registers to be read
$reference-Write	Register to be written
$data	Value array that should be passed to the registers (one value for each register to be written).The amount of values defines the amount of registers to be written.
$dataTypes	Format array for each register to be written (fitting the elements in $data). Possible values: • "WORD" • "INT" • "DINT" • "REAL"

The function returns the read data in the same way as described also in chapter 8.4 of this manual; the arguments that are passed to the function are the same as described in chapter 8.4 and 8.6.

8.9 VALUE CONVERSION

The values that are returned by the read register functions can be converted into PHP format (for the further use within the script), naturally depending on the original data format (please refer to the documentation of the MODBUS device). The following chapters show some examples, where $data always refers to the received byte array (whose length depends on the quantity of read registers, as specified in the corresponding argument).

8.9.1 4 BYTE-VALUES

If the values in the registers of the device are present in 4 byte format (2 registers for each value), they can be „split up" using the following command:

```
$values = array_chunk($data, 4);
```

Now on each value of the array one of the following conversion methods can be applied, depending on the original data format returned by the MODBUS device. See the examples below:

Conversion of the values from REAL to FLOAT:

```
foreach($values as $bytes)
{
   $value = PhpType::bytes2float($bytes);
[...]
}
```

Conversion of the values from DINT to INTEGER:

```
foreach($values as $bytes)
{
    $value = PhpType::bytes2signedInt($bytes);
   [...]
}
```

Conversion of the values from DWORD to INTEGER:

```
foreach($values as $bytes)
{
    $value = PhpType::bytes2unsignedInt($bytes);
   [...]
}
```

8.9.2 2 BYTE-VALUES

If the values in the registers of the device are present in 2 byte format (1 register for each value), they can be „split up" using the following command:

```
$values = array_chunk($data, 2);
```

Now on each value of the array one of the following conversion methods can be applied, depending on the original data format returned by the MODBUS device. See the examples below:

Conversion of the values from INT to INTEGER

```
foreach($values as $bytes)
{
   $value = PhpType::bytes2signedInt($bytes);
[...]
}
```

Conversion of the values from WORD to INTEGER

```
foreach($values as $bytes)
{
    $value = PhpType::bytes2unsignedInt($bytes);
   [...]
}
```

If no conversion, but a formatting of the complete registers as a string format is desired, the following expression can be used:

```
PhpType::bytes2string($data);
```

9 „Sonos“ Library

9.1 INTRODUCTION

The “sonos” library permits to manage one or more SONOS devices (multiroom) within a script. Further information about the SONOS system can be found on www.sonos.com.

9.2 INCLUSION OF THE LIBRARY

In order to include the library, use the following command at the start of the script:

```
include_library(“sonos”);
```

Starting from the next line, the following class can be used within the script:

```
sonosM
```

This class (“SONOS manager”) offers different functions (or “methods”), which permit to load objects from the software and execute operations on them. The class sonosM is used through “static calls”, like for example:

```
sonosM::bind(...);
```

As visible, the static calls start with the name of the class (without the typical variable prefix “$”), followed by two colons and the name of the function to be called; these calls are different from normal object calls (whose names always have the prefix “$”), which use the separator “->” in front of function calls.

9.3 ASSIGNING THE „SONOS“ DEVICE

Before any read / write operations can take place, the class sonosM must be connected to the SONOS device used for the communication. This is done using the following command:

```
sonosM::bind($host);
```

where:

$host IP address of the SONOS device

As long as this function call is not repeated using different arguments, all operations will be sent to the specified SONOS device.

Hint: In order to find the IP address of a SONOS device, you have to install the control software of SONOS on your PC. By opening the information window of the software you will get a list of all configured SONOS devices, together with their IP address:

9.4 CONTROLLING THE „SONOS" DEVICE

As soon as the desired SONOS devices has been connected using the "bind" command, it can be controlled using the following commands:

```
sonosM::stop();         //Stops the playback of the media contents
sonosM::play();         //Starts the playback of the media contents
sonosM::pause();        //Pauses the playback of the media contents
```

Hint: The selection of the media contents that should be used for the playback (playlist / library) must be defined using the control software of SONOS.

Within a running playlist you can use the following commands in order to switch between the tracks:

```
sonosM::next();             //Next track
sonosM::previous();          //Previous track
sonosM::setTrack($number);    //Jump to the track with number $number
```

Furthermore it is possible to forward to a certain playback time (in seconds) within the current track:

```
sonosM::setTime($time);
```

In order to jump back to the beginning of the track you can use:

```
sonosM::rewind();
```

This command has the same effect as using *sonosM::setTime(0)*; .

You can also check the current state of the SONOS device by using the following command:

```
$state = sonosM::getState();
```

The return value of this function can be one of the following:

1 → Playback active (PLAY)

2 → Playback paused (PAUSE)

3 → Playback stopped (STOP)

9.5 VOLUME CONTROL

The volume of a SONOS device can be controlled using the following command...

```
sonosM::setVolume($volume);
```

...where $volume must be an integer value between 0 and 100; furthermore you can also request the current volume of the device:

```
$volume = sonosM::getVolume();
```

Even in this case the return value will be an integer value between 0 and 100.

The volume can also be set to 0 temporarily by using the MUTE command; naturally you can also check if the device cur-rently is in MUTE state:

```
sonosM::setMute();              //Enables the MUTE function
$mute = sonosM::getMute();      //Reads the current MUTE state from the device
```

9.6 PLAYBACK MODE

The following command permits to change the current playback mode:

```
sonosM::setPlayMode($mode);       //Configure playback mode
```

The argument $mode can be one of the following values:

0 → Normal (playback is stopped after the last track)

1 → Repeat (playback starts over from beginning after the last track)

2 → Random (the tracks are played continuously in random order)

9.7 MULTIMEDIA INFORMATION

It is possible to request information about the currently played media content at any time. In order to achieve this, the following command can be used:

```
$info = sonosM::getMediaInfo();       //Read media information of current track
```

The return value is given in form of an array, which contains the following information:

ARTIST	Artist of the track
TITLE	Title of the track
ALBUM	Name of the album
TRACK	Track number
POSITION	Current playback time of the track (in seconds)
DURATION	Complete duration of the track (in seconds)

In order to obtain a specific information from within the array, just use the following commands:

```
$info = sonosM::getMediaInfo();     //Gets the information array
$title = $info[“title”];           //Assigns the title name to the variable $title
$album = $info[“album”];           //Assigns the album name to the variable $album
...
```

9.8 CONFIGURATION EXAMPLE

Your KNXCONTROL device, in combination with the SONOS library, offers a series of example scripts, which explain how the most important functions explained in the past chapters are used in the best way.

Those sample scripts, explained more in detail in chapter 11, can be combined with *Complex objects* in order to create a special SONOS object for the VISUALISATION. The following pages will explain the required steps.

9.8.1 CREATION OF THE COMPLEX OBJECT

As first step you have to create a *Complex object* for every SONOS device that you want to control through the VISUALISATION. The template of the complex object must be set to *MULTIROOM – ZONE*; information about the handling and configuration of *Complex objects* can be found in the *OPTIMA Administrator manual*.

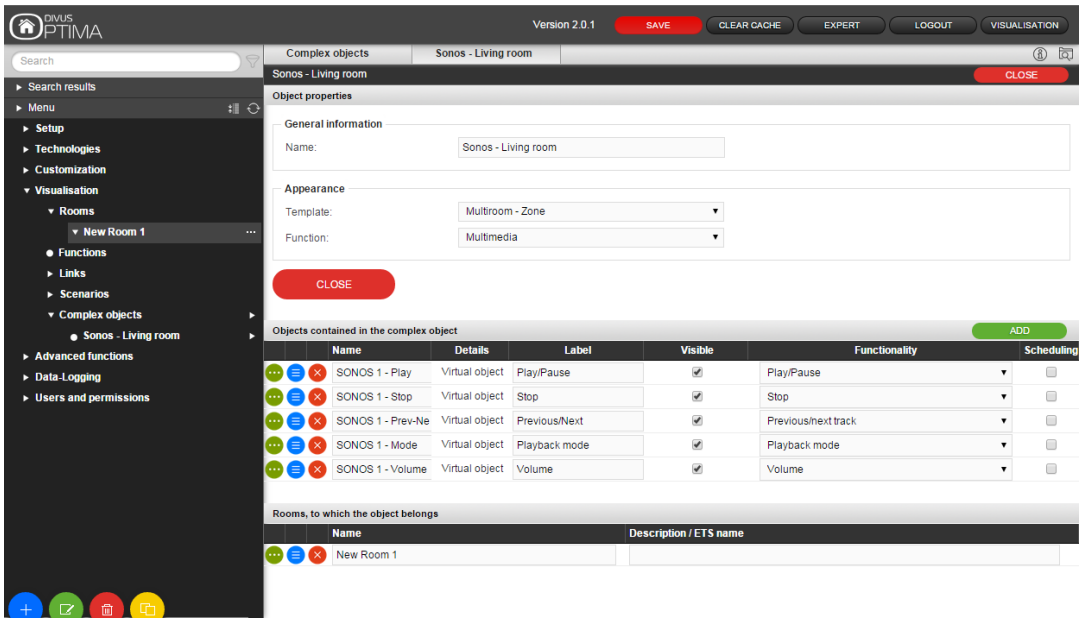
After the creation of the Complex object, please repeatedly click on the "ADD" button in the section *OBJECTS CONTAINED IN THE COMPLEX OBJECT*, in order to add *Virtual Objects* to the *Complex object*. The amount of *Virtual objects* depends on the amount of functionalities that you want to control. The *Complex object* offers the following functions (column FUNCTIONALITY):

- Start / pause playback (PLAY / PAUSE)
- Stop playback (STOP)
- Previous / next track
- Playback mode
- Volume

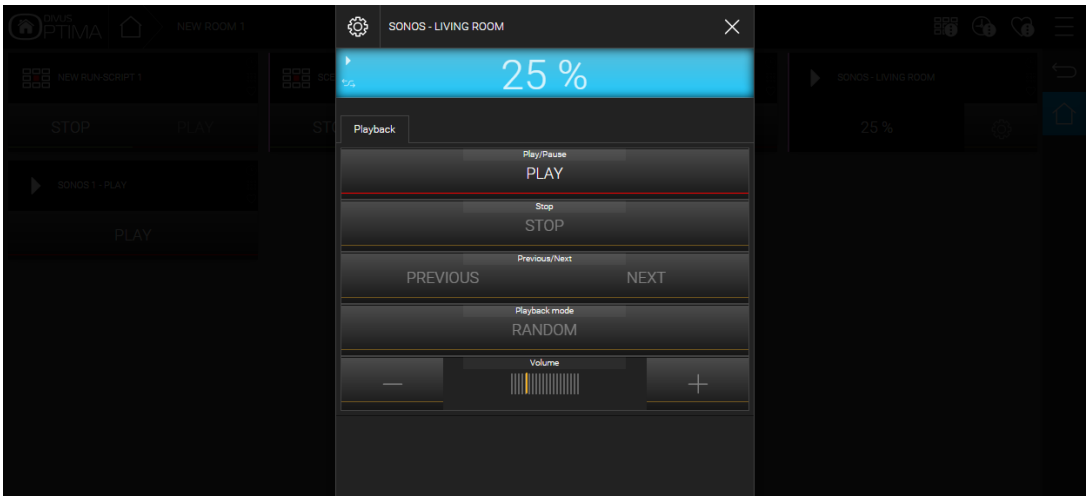
Hint: In order to control additional functions of the SONOS device, please use the „GENERIC" template for the complex object instead of using the "MULTIROOM – ZONE". The configuration of the *generic complex object* is described in the *OPTIMA Administrator manual*.

After the creation of the single sub-objects you have to use the column *Functionality* in order to assign them a function within the complex object and the column *Name* in order to be able to identify the new object. Furthermore, through the column *Label* you can add an additional name which will be used as description of the function inside the complex object in the VISUALISATION.

The following screenshot shows the detail page of the complex object with the newly created virtual objects:



If the complex object is connected to a ROOM, it can be accessed through the VISUALISATION; the objects presents itself in a pop-up like the following:



9.8.2 CREATION AND CONNECTION OF THE RUN-SCRIPTS

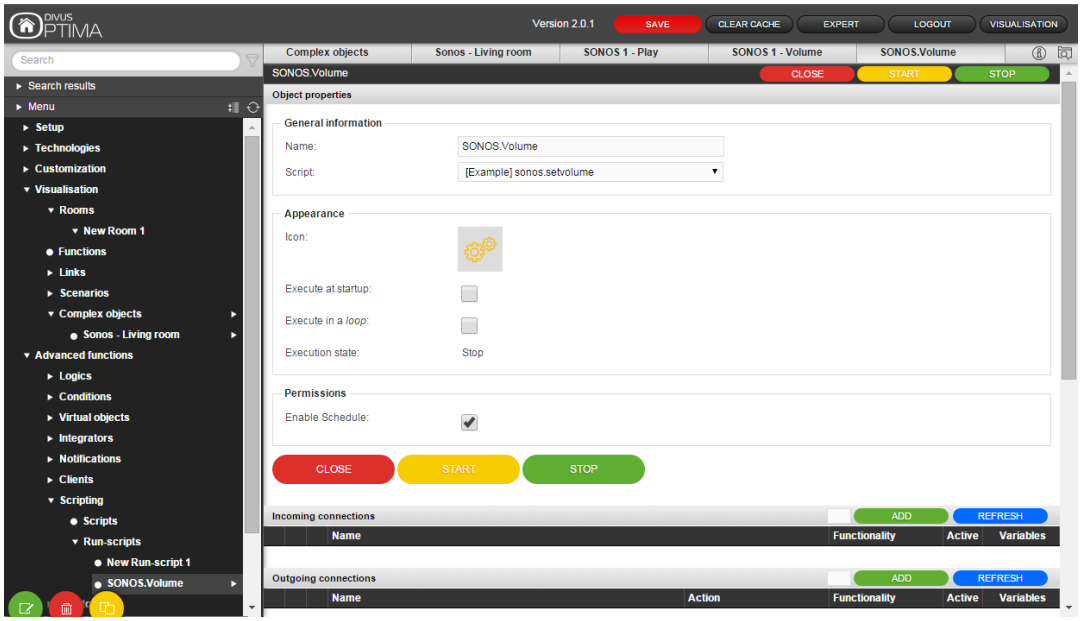
After the creation of the complex object and integration into the VISUALISATION, from a graphical point of view, you have a complete object. Nevertheless, there is no real functionality connected to the different buttons yet. In order to connect the object to a SONOS device, for each function of the complex object a *run-script* must be created, which:

- Must be connected to the script that executes the desired functionality
- Must be connected to the *Virtual object* of the *Complex object*, that should be in control of the desired functionality. The *Virtual object* becomes an INPUT, through which the commands from within the VISUALISATION are passed to the script and are then redirected to the SONOS system.

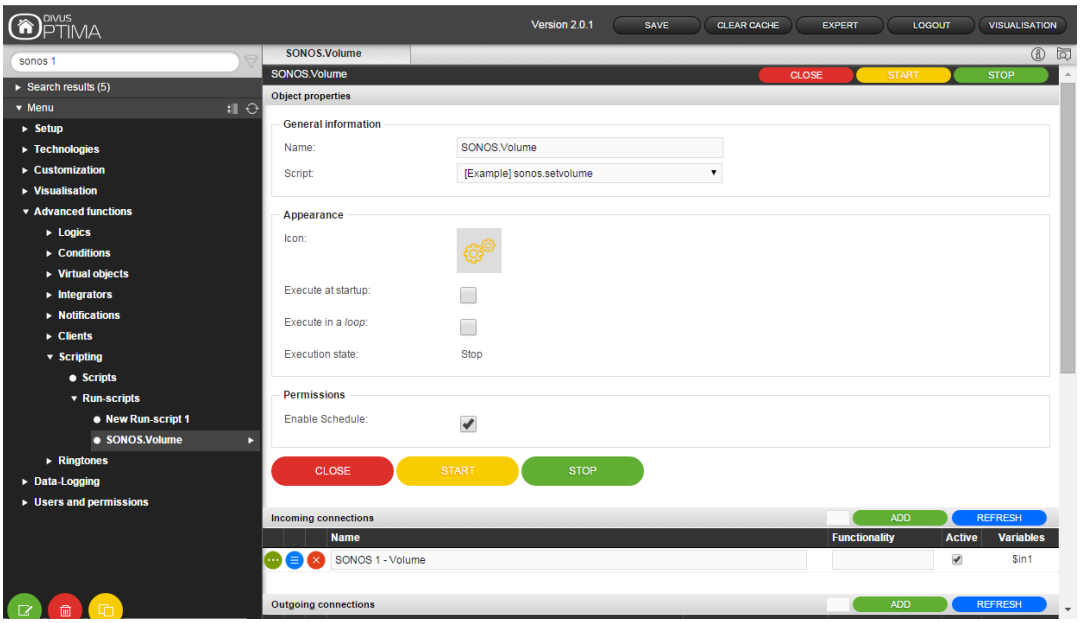
The script samples contained in your KNXCONTROL device already cover all the main functionalities needed to control a SONOS device by using a complex object with template „MULTIROOM – ZONE". Further information about the sample scripts can be found in chapter 11 of this manual.

As an example for the connection of a *Virtual object* and a *run-script,* let's take a look at the volume control of a SONOS device. First of all we create a new run-script for which we want to use the sample script "SONOS.SETVOLUME"'s functionality. We only have to change the IP address inside the script according to our SONOS device's network settings. But, as already mentioned, this is possible only on your personal scripts. So, lookup the script named *sonos.setvolume* in the list of sample scripts under Advanced functions → Scripting → Scripts and klick on the gray clone/copy button in that row. Once done, you will find the copy under your personal scripts list, where you may now change the script's name and – inside the script itself – change the IP address and save the changes.

Since the script neither needs to be executed after startup nor needs to be executed in a loop, the corresponding checkboxes can be ignored.



As next step, we can find the *Virtual object* that previously has been created for the volume control within the *Complex object* (therefore it is important to give a unique *NAME* to the *Virtual object*) and we can connect it via *drag&drop* to the *Incoming connections* of the run-script.



In this way, the value which is set by the user for the volume in the *Complex object* within the *VISUALISATION*, will be passed to the run-script as input value; the run-script will pass the value to the connected script, which will then execute the corresponding function (in this case setVolume()) in order to pass the value to the SONOS device.

10 „Dune“ Library

10.1 INTRODUCTION

The “dune” library permits to manage one or more DUNE-HD devices (multiroom/multimedia) within a script. Further information about the DUNE system can be found on http://dune-hd.com/.

10.2 INCLUSION OF THE LIBRARY

In order to include the library, use the following command at the start of the script:

```
include_library(“dune”);
```

Starting from the next line, the following class can be used within the script:

```
duneM
```

This class (“DUNE manager”) offers different functions (or “methods”), which permit to load objects from the software and execute operations on them. The class duneM is used through “static calls”, like for example:

```
duneM::bind(...);
```

The static calls start with the name of the class (without the typical variable prefix “$”), followed by two colons and the name of the function to be called; these calls are different from normal object calls whose names always have the prefix “$” and which use the separator “->” in front of function calls.

10.3 ASSIGNING THE „DUNE“ DEVICE

Before any read / write operations can take place, the class duneM must be connected to the DUNE-HD device used for the communication. This is done using the following command:

```
duneM::bind($host);
```

Where:

$host IP ADDRESS OF THE DUNE DEVICE

As long as this function call is not repeated using different arguments, all operations will be sent to the specified DUNE device.

10.4 GENERAL COMMANDS OF THE „DUNE" DEVICE

The communication with the DUNE device is done using network telegrams, which have to comply to the directives of the following protocol specification:

http://dune-hd.com/support/ip_control/dune_ip_control_overview.txt

The exact specification furthermore depends on the firmware version of the used DUNE device.

The "dune" library offers the following, general command:

```
$result = duneM::call($cmdString);
```

Using the argument $cmdString the desired command can be passed to the DUNE device as described in the documentation; simply use the part after the character "?". Example: if in the documentation the command is...

```
http://10.0.0.1/cgi-bin/do?
cmd=start_file_playback&media_url=nfs://10.0.0.1:/VideoStorage:/SomeFolder/file.mkv
```

... the function of the library must be used in the following way:

```
duneM::call("cmd=start file playback&media url=nfs://10.0.0.1:/VideoStorage:/SomeF
older/file.mkv");
```

If the execution of the command was successful, the function returns the value _SCRIPT_RESULT_NOERROR; if instead there has been an error, the return value is _SCRIPT_RESULT_USERERROR.

The return value of the most recently executed command will be stored in the following global variable:

```
$_duneLastResponse
```

This variable is an object which stores all the information received by the DUNE device in its attributes. The result of the executed action ("ok" if no error, otherwise the corresponding error message) can be requested by using the following input:

```
$_duneLastResponse->command_status
```

10.5 CONTROL OF THE „DUNE" DEVICE

In order to simplify the handling of the scripts for the control of DUNE devices, the library offers a series of functions that already contain the most common multimedia controls. This makes the usage of the single scripts easier and helps you keeping an overview.

For example, in order to play a certain multimedia content, depending on the type of content one of the following commands can be used:

```
duneM::playFile($url);           //Playback of a general movie file
duneM::playDVD($url);            // Playback of a DVD or DVD image file
duneM::playBluRay($url);         // Playback of a BluRay or BluRay image file
duneM::playList($url,$track);    // Playback of a playlist, with optional parameter
for specifying the first track to be played
```

The argument $url must contain the path to the multimedia source (CD, DVD, BLU-RAY) or the file / playlist that should be reproduced. The path must be specified in the format requested by the DUNE protocol. Some examples:

```
duneM::playFile("nfs://10.0.0.1:/VideoStorage:/SomeFolder/file.mkv");
duneM::playDVD("smb://10.0.0.1/VideoStorage/SomeFolder/DVDFolder");
duneM::playDVD("storage_name://MyHDD1/SomeFolder/dvd_image.iso");
duneM::playBluRay("nfs://10.0.0.1:/VideoStorage:/SomeFolder/BlurayFolder");
duneM::playBluRay("nfs://10.0.0.1:/VideoStorage:/SomeFolder/bluray_image.iso");
```

The following commands can be used to switch the DUNE device at any time either to a "blackscreen" (stopping the cur-rent playback) or the main screen of the device:

```
duneM::blackScreen();
duneM::mainScreen();
```

For switching the device into standby, the following command can be used:

```
duneM::standby();
```

10.6 EMULATION OF THE REMOTE CONTROL

The library also offers the possibility to emulate the commands of the DUNE remote control. Therefore, the following command exists:

```
duneM::irCode($code);
```

The argument $code in this case must contain the IR code, which stands for the desired action; the IR codes are docu-mented on the following web page:

http://dune-hd.com/support/rc

Example: in order to emulate the button "1" of the remote control, the following command must be used:

```
duneM::irCode("F40BBF00");
```

In order to simplify the emulation of the remote control within a script, the library offers a second function, which instead of the IR codes accepts the description of the button of the remote control:

```
duneM::irButton($button);
```

In this case the button "1" can simply be emulated through the following command:

```
duneM::irButton("1");
```

The following table shows an overview of the arguments supported by the second function. Please make sure to input the values correctly (e.g. capitalization). Furthermore be sure to input the arguments in quotation marks, since they have to be passed as STRINGS:

EJECT	MUTE	MODE	POWER	A	B	C	D
1	2	3	4	5	6	7	8
9	0	CLEAR	SELECT	V+	V-	P+	P-
SETUP	UP	DOWN	LEFT	RIGHT	ENTER	RETURN	INFO
PLAY	PAUSE	PREV	NEXT	STOP	REW	FWD	REC

Further information regarding the available buttons and supported functionalities can be found in the online documentation of DUNE.

10.7 PLAYBACK CONTROL

The volume of the DUNE device can be controlled using the following function:

```
duneM::setVolume($volume);
```

The argument $volume must be an integer value between 0 and 100; furthermore the device can be muted (value 1) or unmuted (value 0) using the following command:

```
duneM::setMute($mute);
```

Finally, you can also start and pause the playback of the current multimedia content by using the following two com-mands:

```
duneM::play();
duneM::pause();
```

11 „Messages“ Library

11.1 INTRODUCTION

The “messages” library permits to send out onscreen and mail notifications directly through a script, without having to create the normally needed objects within the VISUALISATION.

11.2 INCLUSION OF THE LIBRARY

In order to include the library, use the following command at the start of the script:

```
include_library(“messages”);
```

Starting from the next line, the following class can be used within the script:

```
msgM
```

This class (“MESSAGE manager”) offers different functions (or “methods”), which permit to load objects from the software and execute operations on them. The class msgM is used through “static calls”, like for example:

```
msgM::sendVideoMsg(...);
```

As visible, the static calls start with the name of the class (without the typical variable prefix “$”), followed by two colons and the name of the function to be called; these calls are different from normal object calls (whose names always have the prefix “$”), which use the separator “->” in front of function calls.

11.3 SENDING OUT ONSCREEN NOTIFICATIONS

In order to create and send out a new onscreen notification within a script, you can use the following command:

```
msgM::sendVideoMsg($msg,$level);
```

Where:

$msg	TEXT / CONTENTS OF THE NOTIFICATION
$level	PRIORITY LEVEL; POSSIBLE VALUES:
	0 Alarm (Standard)
	1 Warning
	2 Information

Example:

```
msgM::sendVideoMsg("Hello World!",2);
```

Naturally it is also possible to pass object values within the notification, for example the value of an object connected as INPUT of the run-script:

```
msgM::sendVideoMsg("Input value" . input() . " received!",2);
```

If the notification was sent out with success, the function will throw out the return value _SCRIPT_RESULT_NOERROR; in case of errors the return value will contain the error description.

11.4 SENDING OUT MAIL NOTIFICATIONS

In very similar way also mail notifications can be sent out using the following command:

```
msgM::sendEMailMsg($msg,$to,$subject,$cc,$bcc);
```

where:

$msg	Text / contents of the mail
$to	Mail address(es) of the receiver (please divide multiple addresses with ";")
$subject	Subject of the mail
$cc	Mail addresses in copy

$bcc	Mail addresses in copy (hidden)

Also this function, if the notification was sent out with success, will throw out the return value *_SCRIPT_RESULT_NOERROR*; in case of errors the return value will contain the error description.

12 Examples

12.1 INTRODUCTION

This chapter presents and explains the single sample scripts preinstalled in your KNXCONTROL device. They can be used as starting point for the creation of customized scripts.

12.2 LOGIC.AND

This sample script executes a logical AND connection on all the objects connected to the INPUT of the run-script that integrates this sample script. Please find the PHP code below (some of the comments have been left out to make the code easier readable):

```
include_library("surrounding");

//Initializing logic result
$result = true;

//Scanning parent relations
foreach($me->parentRelations as $parentRelation)
{
    $parent = $parentRelation->parent;
    $parent_value = $parent->value;

    //Calculating new logic result by putting parent value in AND with previous
result
    $result = $result && (!empty($parent_value));

    //If result is no longer true, skipping: no need for further calculation, it
won't never be true again!
    if(!$result)  break;
}

//Sending result to output - In this way, runner will pass it to child (active event)
output($result);
```

This example is very similar to the one shown in chapter 6.4 of this manual: even here as first step a control of all PARENT relations (objects in the INPUT of the run-script) is done and the value of each of these objects is read; those values are then used in order to evaluate the AND connection sequentially (between the previous object and the next one).

If the result gets FALSE once, the loop is interrupted, since a logical AND connection in this case can never become TRUE again.

The calculated value is afterwards returned by calling the function output().

12.3 LOGIC.OR

In the same way as the previous script, this sample script executes a logical OR connection on all the objects in the INPUT. The main differences to the previous script are shown in red:

```
include_library("surrounding");

//Initializing logic result
$result = false;

foreach($me->parentRelations as $parentRelation)
{
    $parent = $parentRelation->parent;
    $parent_value = $parent->value;

    //Calculating new logic result by putting parent value in OR with previous result
    $result = $result || (!empty($parent_value));

    //If result is true, skipping: no need for further calculation, it won't never
be false again!
    if($result)  break;
}

//Sending result to output - In this way, runner will pass it to child (active event)
output($result);
```

In this case, the loop is automatically interrupted as soon as the value of one of the objects in the input is different from 0 (zero); in this moment the logical OR connection can't become FALSE again and the code can be interrupted.

12.4 SERIAL.WRITER_GENERIC

This sample script shows how to write data on the serial interface; as message the INPUT value of the run-script is used. Also in this case some comments have been removed from the sample code in order to keep it readable:

```
include_library("serial");

//Uncomment following lines if you need to change default serial settings
//$serialM->deviceSet("/dev/ttyS4");     //ATTENTION: change it at your own risk! This is default RS232 port of KNXSERVER
//$serialM->confBaudRate(19200);    //Change    baudrate    -    Allowed    values: 110,150,300,600,1200,2400,4800,9600,19200,38400,57600,115200
//$serialM->confParity("none");           //Change   parity   -   Allowed   values: "none","odd","even"
//$serialM->confCharacterLength(8);//Change character length
//$serialM->confStopBits(1);      //Change stop bits
//$serialM->confFlowControl("none");//Change   flow   control   -   Allowed   values: "none","rts/cts","xon/xoff"

//Opening serial port
$serialM->deviceOpen();

//Initializing message as input value
$msg = input();

//Sending message to serial port, by adding a trailing return at the end (new line)
$serialM->sendMessage($msg . "\n");

//Closing serial port
$serialM->deviceClose();

//Sending input value to output - In this way, value is passed to any eventual runner's child and set as its value
output(input());
```

As you may notice, above the command to open the serial port, all commands to change the communication parameters of the interface are present in the script. So if you need to change those parameters, you can just uncomment the desired functions (removing the prefix "//") and set the desired value, as described in chapter 7.3.

The value that is sent to the interface is the INPUT value (called through the function input()) of the run-script, followed by the characters to jump to a new line (can be adapted for your own needs).

At termination of the write operation the interface is closed and the script will return the sent value by passing it to the function output(); in this way the run-script – in case it should also be present in a page of the visualisation – can also show the sent value. This step is optional and has nothing to do with the write operation itself; therefore you are free to modify or remove the argument of this command.

12.5 DEMUX.STATUSBYTE

This sample script executes a "demultiplexing" operation on the INPUT value of the run-script. It is necessary that the INPUT value is a 1 byte value (numeric value between 0 and 255). This value is split up by the script into 8 values of 1 bit, which consequently are available in the binary format.

Hint: This script is really useful when you want to handle so called "status bytes", which are used by different KNX devices in order to send their current state to the bus (therefore the name of this sample script). The script „extracts" this value and permits to assign the single bit values to 8 objects connected to the output of the run-script, allowing them to show the state of the main object.

This script requires that the connected run-script has connected 8 objects in its OUTPUT section. To each of those objects the corresponding bit value (resulting from „demultiplexing" the input value) is assigned, in the order they appear within the outputs section of the run-script.

```
include_library("surrounding");

//Converting input value to binary
$statusByte = str_pad(decbin(input()),8,'0',STR_PAD_LEFT);

//Scanning childs up to 8th, setting corresponding demuxed value
for($i=0;$i<8;$i++)
{
  //Initializing output name into a temporary variable
  $outName = "out" . ($i+1);

  //If variable is not set, exiting loop - It means we don't have enough children
associated to the run-script
  if(!isSet($$outName)) break;

  //Setting value to child
  $$outName->set($statusByte[$i] ? 1 : 0);
}

//Returning an empty string - In such a way, environment won't call children again
(since we already did it!)
output();
```

The following line executes the "demultiplexing" operation:

```
$statusByte = str_pad(decbin(input()),8,'0',STR_PAD_LEFT);
```

The obtained numeric value is firstly converted to binary code (missing "zeros" at the beginning are added automatically, in order to get a fixed length of 8 bit). As next step, the objects connected as outputs of the run-script (from 1 to 8) are assigned to a variable using a loop:

```
$outName = "out" . ($i+1);
```

Then the script checks if the output is really existing; if no object can be detected (e.g. because not connected in the run-script), the loop is interrupted:

```
if(!isSet($$outName)) break;
```

If 8 valid objects are present in the output, a "set" command is executed, which passes the calculated bit value (either 0 or 1) to the output:

```
$$outName->set($statusByte[$i] ? 1 : 0);
```

Hint: The PHP language permits to use the value of variable as a variable, too, like used at...
$$outName
... in the script. Instead of writing the value of $outName (= "out1", "out2" etc.) down, like...
$out1, $out2 etc...
... it is also possible to use directly $$outName, which in each iteration of the loop corresponds to the referenced object in the output of the run-script (check out chapter 6.3 for the explanation of $out).

At the end the script is terminated using the following function:

```
output();
```

This means that no value is returned by the script and therefore prevents that the objects in the output of the run-script, which have already been commanded directly from within the script, are executed again.

12.6 MATH.SUM

This sample script calculates the SUM of the values of all objects connected in the input of the run-script. Find below the sample code (the comments have been left out to make the code easier readable):

```
include_library("surrounding");

$sum = 0;

foreach($me->parentRelations as $parentRelation)
{
    $parent = $parentRelation->parent;
    $parent_value = $parent->value;

    $sum = $sum + floatval($parent_value);
}

output($sum);
```

Also this example is very similar to the one shown in chapter 6.4 of this manual: as first step a control of all PARENT relations is done and the value of each of these objects is read; those values are then used in order to calculate the sum sequentially.

The calculated value is afterwards returned by calling the function output().

Note: The following version of the script is based on the same concept, but filters out some hidden objects which were introduced in newer versions of Optima and could cause the above script to fail. The added lines are bold.

```
include_library("surrounding");

$sum = 0;

foreach($me->parentRelations as $parentRelation)
{
    $parent = $parentRelation->parent;
    $parent_value = $parent->value;
    $parent_type = $parent->type;

     if ((!empty($parent_value))&&($parent_type!="LOGIC"&&$parent_type!="GROUP")) {
        $sum = $sum + floatval($parent_value);
    }
}

output($sum);
```

12.7 MATH.PRODUCT

This sample script calculates the PRODUCT of the values of all objects connected in the input of the run-script. Find below the sample code (the comments have been left out to make the code easier readable):

```
include_library("surrounding");

$product = 1;

foreach($me->parentRelations as $parentRelation)
{
    $parent = $parentRelation->parent;
    $parent_value = $parent->value;

    $product = $product * floatval($parent_value);
}

output($product);
```

The script is identical with the previous example, with the difference (beneath the "*" sign instead of "+") that the result must be initialized with value 1 instead of 0.

Also in this case the calculated value is returned by calling the function output().

See 12.6 (2nd script) for a similar script which filters out some hidden object types – the same procedure might be necessary here.

12.8 MODBUS.READCOILS

This sample script reads a series of coils (binary in/outputs) from a MODBUS SLAVE device and can consequently command an object connected as OUTPUT of the run-script:

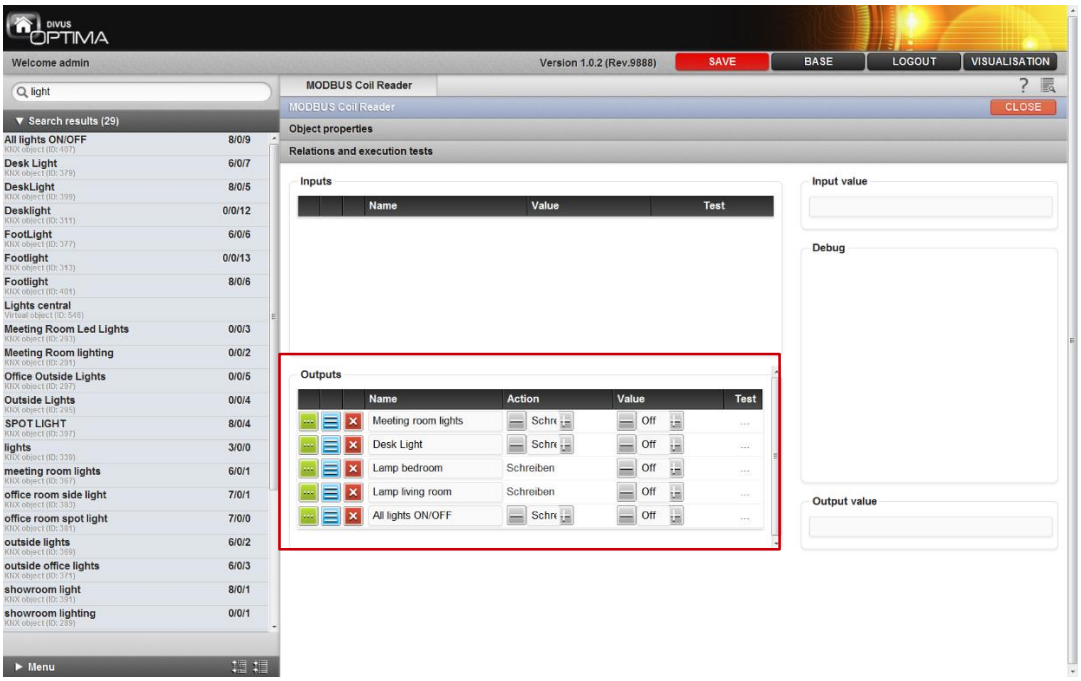
```
include_library("surrounding");

$host = "192.168.0.71";   //Change this according with MODBUS SLAVE IP address
$protocol = "TCP";        //Same as above
$start = 0;               //First coil to read

$number = count($me->childRelations);
if($number>0)
{
 include_library("modbus");

 modbusM::bind($host,$protocol);

 $values = modbusM::readCoils(0, $start, $number);

 for($i=0;$i<count($values);$i++)
 {
  $value = !empty($values[$i]) ? 1 : 0;
  debug("Coil " . ($start + $i) . " has value $value",true);

  $childName = "out" . ($i+1);
  if(is_object($$childName))
  {
   if($$childName->value == $value) continue;

   $$childName->set($value);
   debug("Object " . $$childName->name . " set to $value",true);
  }
 }
}

output();
```

As you can easily notice, the first part of the script is used to initialize the parameters necessary for the communication with the MODBUS SLAVE device:

```
$host = "192.168.0.71";   //Change this according with MODBUS SLAVE IP address
$protocol = "TCP";        //Same as above
$start = 0;               //First coil to read
```

Besides IP address and protocol it is also important to specify the MODBUS address of the first coil to be read. Please refer to the documentation of your MODBUS device in order to find out the correct value (please keep in mind that the command to read out the values already contains an internal offset, generally "10000").

The script has been designed to be launched from a run-script with the same amount of objects connected to its output as the amount of coils that should be read from the MODBUS SLAVE device:



After the start, the script checks the amount of objects present in the output of the run-script ...

```
$number = count($me->childRelations);
```

... and then reads out (after controlling if at least one object is present and if the "modbus" library has been included) the same amount of coils from the MODBUS SLAVE device:

```
$values = modbusM::readCoils(0, $start, $number);
```

Now the array with the results (that contains the same amount of elements as the amount of objects in the output) is passed through a loop, the current value (either "1" or "0", depending on the read value) is written into a variable...

```
$value = !empty($values[$i]) ? 1 : 0;
```

... just like the corresponding output is written to a variable:

```
$childName = "out" . ($i+1);
```

As already explained in chapter 9.5 the following expression is used to access the output directly:

```
$$childName
```

After a short check if the output is really existing...

```
if(is_object($$childName))
```

... the script controls if the object connected as output already has the same state as the value received from the MODBUS device ($value), since in this case no action would be necessary:

```
if($$childName->value == $value) continue;
```

If those values are different, the value read from the MODBUS device is set to the output object:

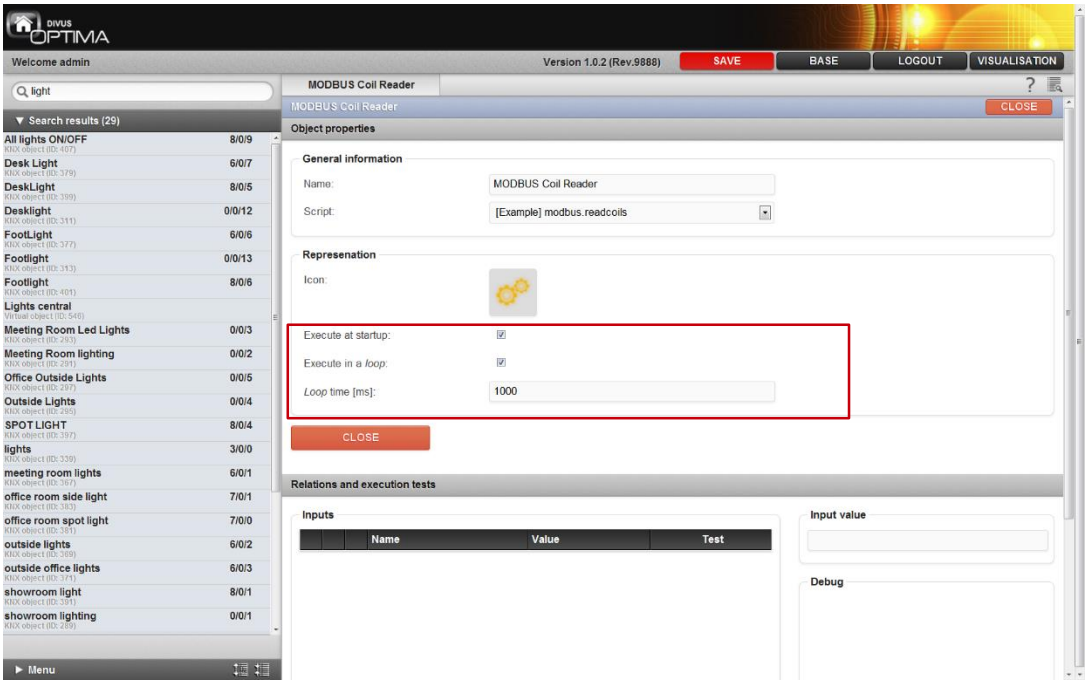
```
$$childName->set($value);
```

At conclusion, the script is terminated without passing an OUTPUT value:

```
output();
```

Also here this last step is very important, since otherwise the output objects would be controlled again and the previously set values would be overwritten.

Typically a run-script connected to such kind of scripts will be started immediately after the start of the software and should be running continuously in background:



The loop time defines the duration between one and the next execution of the script, in this case the frequency, how often the coils of the MODBUS devices are read ("polling" time). Therefore, this value also defines the "reaction time", in which status changes on the MODBUS devices will be recognized and elaborated by the KNXCONTROL device.

Hint: When using too small values for the loop time (< one second), you risk to overload your KNXCONTROL device, above all when many coils must be read (the same goes for similar scripts which read out the registers of MODBUS devices).

12.9 MODBUS.READREGISTERS

This sample script is based on the same concept as the previous one, but instead of reading out coils it reads registers from the MODBUS SLAVE device. For an easier reading, the main differences to the previous script are highlighted in red:

```
include_library("surrounding");
$host = "192.168.0.71";  //Change this according with MODBUS SLAVE IP address
$protocol = "TCP";       //Same as above
$start = 0;              //First register to read
$dataLength = 2;         //Bytes for each data - Typically 2 or 4 bytes

$number = count($me->childRelations);
if($number>0)
{
 include_library("modbus");

 modbusM::bind($host,$protocol);

 $data = modbusM::readRegisters(0, $start, $number);

 $values = array_chunk($data, $dataLength);
 for($i=0;$i<count($values);$i++)
 {
  $value = PhpType::bytes2signedInt($values[$i]);
  debug("Register " . ($start + $i) . " has value $value",true);

  $childName = "out" . ($i+1);
  if(is_object($$childName))
  {
   $$childName->set($value);
   debug("Object " . $$childName->name . " set to $value",true);
  }
 }
}
output();
```

For this script it is also necessary to specify in which format (2 or 4 byte) the data should be read from the MODBUS – device and sequentially be passed to the output objects:

```
$dataLength = 2;         //Bytes for each data - Typically 2 or 4 bytes
```

With this information, the resulting data array can be split up...

```
$values = array_chunk($data, $dataLength);
```

... and in the connection passed value per value to the objects connected as outputs of the run-script. Depending on the data format received from the MODBUS device it could be possible that a corresponding conversion of the data is made, like for example:

```
$value = PhpType::bytes2signedInt($values[$i]);
```

Further information regarding the various conversion methods can be found in the chapter 8.9 of this manual.

Hint: For keeping the example simple, the script doesn't contain a control of the value of the output object before sending out the value received via MODBUS. In any case it is recommended to insert this control – just like in the script reading out the coils – since in this way it can be prevented that values are continuously written to the bus, even if not necessary (most important when the script is executed in a loop).

Also for this sample script it makes sense to be executed at startup of the software and to run continuously in background (in a loop).

12.10 MODBUS.WRITECOILS

This sample script writes a series of coils to a MODBUS SLAVE device, depending on status changes of the objects connected as inputs of the run-script:

```
include_library("surrounding");

$host = "192.168.0.71";   //Change this according with MODBUS SLAVE IP address
$protocol = "TCP";        //Same as above
$start = 0;               //First coil to write

$number = count($me->parentRelations);
if($number>0)
{
 include_library("modbus");

 modbusM::bind($host,$protocol);

 $values = array();

 foreach($me->parentRelations as $parentRelation)
 {
  $parent = $parentRelation->parent;
  $value = $parent->value;

  $values[] = (!empty($value) ? TRUE : FALSE);
 }

 $result = modbusM::writeCoils(0, $start, $values);
 debug("Written coils to modbus slave: " . implode(",",$values));
}

output();
```

As already seen in the last chapters, the first part of the script is used to configure the communication parameters of the MODBUS SLAVE device:

```
$host = "192.168.0.71";   //Change this according with MODBUS SLAVE IP address
$protocol = "TCP";       //Same as above
$start = 0;             //First coil to write
```

This script also requires the number of the first coil to be written ($start). The amount of coils to be written is determined through the amount of objects connected as input of the run-script:

```
$number = count($me->parentRelations);
```

After the inclusion of the "modbus" library and the control, if at least one object is present in the input section, the script creates a value array using the current values of the input objects...

```
foreach($me->parentRelations as $parentRelation)
 {
  $parent = $parentRelation->parent;
  $value = $parent->value;

  $values[] = (!empty($value) ? TRUE : FALSE);
 }
```

... and sends them to the MODBUS device:

```
$result = modbusM::writeCoils(0, $start, $values);
```

Also in this case the script doesn't return any value and is terminated after the write operation.

12.11 SONOS.GENERIC

The following script shows a short demo application for the control of a SONOS device:

```
include_library("sonos");

sonosM::bind("192.168.0.150");

sonosM::play();
debug("Now SONOS is playing...",true);

sleep(5);

sonosM::setVolume(50);
debug("Now SONOS volume is 50%...",true);

sleep(5);

sonosM::next();
debug("Skipped to next SONOS track...",true);

sleep(5);

sonosM::pause();
debug("Now SONOS is in pause...",true);

output();
```

After connecting to a SONOS device with IP address 192.168.0.150, the script sends different commands to the device, each separated by 5 seconds of break.

12.12 SONOS.SETVOLUME

This script sets the volume of a SONOS device depending on the received input value (normally the value of an object connected as INPUT of the run-script that is executing the current script):

```
include_library("sonos");
sonosM::bind("192.168.0.150");
$volume = input();
sonosM::setVolume($volume);
output();
```

Hint: This script is compatible as well with the "VOLUME" function for KNX objects as also with the volume function of the complex object "MULTIROOM - ZONE".

12.13 SONOS.GETVOLUME

This script gets the current volume value from the SONOS device and returns it as output value (and therefore normally passes it to an object connected as OUTPUT of the run-script that is executing the current script):

```
include_library("sonos");
sonosM::bind("192.168.0.150");
$volume = sonosM::getVolume();
output($volume);
```

12.14 SONOS.PLAYPAUSE

This script starts the multimedia playback (PLAY) of the SONOS device or pauses the playback (PAUSE), depending on the value received as input (1 or 0):

```
include_library("sonos");
sonosM::bind("192.168.0.150");
$cmd = input();
if(intval($cmd)==1)
 sonosM::play();
else
 sonosM::pause();

output();
```

Hint: This script is compatible also with the "PLAY/PAUSE" function for KNX objects and with the play/pause function of the *Complex object MULTIROOM - ZONE*.

12.15 SONOS.PREVNEXT

This script jumps to the next or previous track of the playlist of the SONOS device, depending on the value received as input (1 or 0):

```
include_library("sonos");
sonosM::bind("192.168.0.150");
$cmd = input();
if(intval($cmd)==1)
 sonosM::next();
else
 sonosM::previous();
output();
```

Hint: This script is compatible with the "PREVIOUS/NEXT" function for KNX objects as well as with the previous/next track function of the *Complex object MULTIROOM - ZONE*.

12.16 SONOS.SETPLAYMODE

This script changes the playback mode of a SONOS device, depending on the received input value:

```
include_library("sonos");
sonosM::bind("192.168.0.150");
$mode = input();
sonosM::setPlayMode($mode);
output();
```

Hint: This script is compatible with the "PLAYBACK MODE" function for KNX objects as well as also with the playback mode function of the *Complex object MULTIROOM – ZONE*.

12.17 SONOS.GETINFO

This script requests information about the currently played track from the SONOS device and can pass this information to objects connected as Outgoing connections of the run-script (the order of how the values are passed from within the script also determines the order in which the corresponding objects have to be connected in the output section of the run-script):

```
include_library("surrounding");
include_library("sonos");
sonosM::bind("192.168.0.150");
$info = sonosM::getMediaInfo();

if(isSet($out1) && ($out1->value != $info["artist"]))   $out1->set($info["artist"]);
if(isSet($out2) && ($out2->value != $info["title"]))    $out2->set($info["title"]);
if(isSet($out3) && ($out3->value != $info["album"]))    $out3->set($info["album"]);
if(isSet($out4) && ($out4->value != $info["Track"]))    $out4->set($info["Track"]);
if(isSet($out5) && ($out5->value != $info["position"])) $out5->set($info["position"]);
if(isSet($out6) && ($out6->value != $info["duration"])) $out6->set($info["duration"]);

output();
```

12.18 DUNE.ONOFF

This script turns a DUNE device either off or on, depending on the value received as input (0 or 1):

```
include_library("dune");
duneM::bind("192.168.0.150");
$cmd = input();
if(intval($cmd)==1)
 duneM::mainScreen();
else
 duneM::standby();

output();
```

12.19 DUNE.PLAYPAUSE

This script turns the multimedia playback of a DUNE device on (PLAY) or pauses it (PAUSE), depending on the value received as input (0 or 1):

```
include_library("dune");
duneM::bind("192.168.0.150");
$cmd = input();

if(intval($cmd)==1)
 duneM::play();
else
 duneM::pause();

output();
```

Hint: This script is compatible as well with the "PLAY/PAUSE" function for KNX objects as also with the play/pause function of the complex object "MULTIROOM - ZONE".

12.20 DUNE.SETVOLUME

This script sets the volume of a DUNE device depending on the received input value:

```
include_library("dune");
duneM::bind("192.168.0.150");

$volume = input();
duneM::setVolume($volume);

output();
```

Hint: This script is compatible as well with the "VOLUME" function for KNX objects as also with the volume function of the complex object "MULTIROOM - ZONE".

12.21 DUNE.IRCOMMAND

This script emulates the pressure of a button on the remote control of the DUNE system, depending on the received input value:

```
include_library("dune");
duneM::bind("192.168.0.150");

$button = input();
duneM::irButton($button);

output();
```

12.22 DEWPOINT

This script calculates the dew point depending on a TEMPERATURE and a relative HUMIDITY value, which need to be connected as INPUT objects of the run-script. In the example below we assume that the temperature has been added as first input and the humidity as second input value:

```
include_library("surrounding");

$T = floatval($in1->value);
$H = floatval($in2->value);

$D = round(pow(($H/100),(1/8)) * (112 + (0.9 * $T)) + (0.1 * $T) - 112,1);
output($D);
```

Hint: The calculation is done based on the approximate formula explained on the following page: *http://en.wikipedia.org/wiki/Dew_point*

13 Appendix

13.1 FUNCTION – OBJECT TYPES

The table on the following page shows a list of all the function types of the objects present in the OPTIMA inteface. Those are used by the communication service in order to execute all kind of actions on the low level part of the VISUALISATION. Normally this kind of object types in the software is referred as "TYPE".

Beneath a description, the table contains also all executable actions for each object type; these can be called using the following command:

```
objM::objPerformOperation(ID,OPERATION,VALUE);
```

Details regarding the correct usage of this function can be found within chapter 5.5. of this manual.

TYP ("TYPE")	DESCRIPTION	ACTIONS
VIRTUALOBJECT	Virtual object	SETVALUE → Sets the value of an object
SCENARIO	Scenario	EXECUTE → Executes the scenario STOPEXECUTION → Interrupts the scenario
LOGIC	Logic	EVALUATE → Evaluates the logic
CONDITION	Condition	CALCULATE → Calculates the condition
CLIENTBROWSER	Client	REDIRECT → Executes a page jump on the client device VOIPCALL → Calls the corresponding client device
NOTIFY_VIDEO	Onscreen notification	INSERT → Inserts an onscreen notification
NOTIFY_EMAIL	Mail notification	SENDMAIL → Sends out a mail notification
INTEGRATOR	Integrator	INTEGRATE → Refreshes the calculation of the integrator RESET → Resets the integrator back to 0
EIBOBJECT	KNX object	SETVALUE → Sends a value to the group address of the KNX object

		GETVALUE → Sends a status request to the group address of the KNX object
CSCMD	Run-script	RUN → Executes the script QUIT → Interrupts the script
USER	User	REDIRECT → Executes a page jump on all devices on which the user is currently online VOIPCALL → Calls all the client devices on which the user is currently online
USERGROUP	User group	
USERPERMISSION	Permission	
CONTAINER	Complex object	
CAMERA	IP camera	
PBXELEMENT	VoIP participant External unit Call group	VOIPCALL → Calls the corresponding VoIP object

13.2 WEB – OBJECT TYPES

The following table shows a list of all the web types of the objects present in the OPTIMA interface. Those object types are used by the web interface for the graphical representation of the previously explained function types and in the software are referred as "PHPCLASS".

For each web type – if present – the corresponding function type (or "TYPE", see previous chapter) is listed; it is also possible that more than one PHPCLASS is connected to the same "TYPE". If instead the field "TYPE" is empty, the corresponding PHPCLASS has no connection to the communication service of the software and is therefore used purely for the VISUALISATION itself (like for example ROOMS, LINKS, ...).

WEB TYPE ("PHPCLASS")	DESCRIPTION	FUNKTION TYPE ("TYPE")
dpadObject	General object*	
dpadGroup	Room (or folder/ container / groups)	
dpadUrl	Link	
dpadSysCmd	System command*	SYSCMD
dpadSysCmdWait	Wait command of a scenario	SYSCMD
dpadUser	User	USER
dpadUserGroup	User group	USERGROUP
dpadUserPermission	Permission of a user group	USERPERMISSION
dpadTriggerObject	Trigger for database changes*	TRIGGEROBJECT
dpadContainer	Complex object	CONTAINER
dpadVirtualObject	Virtual object	VIRTUALOBJECT
dpadScenario	Scenario	SCENARIO
dpadPbxExtension	VoIP participant	PBXELEMENT
dpadPbxTrunk	Phone line	
dpadPbxQueue	Call group	
dpadPbxDoorOpener	External unit	
dpadLogic	Logic	LOGIC
dpadCondition	Condition	CONDITION
dpadClientBrowser	Client	CLIENTBROWSER
dpadNotifyVideo	Onscreen notification	NOTIFY_VIDEO
dpadNotifyEmail	Mail notification	NOTIFY_EMAIL

dpadIntegrator	Integrator	INTEGRATOR
dpadEibObject	KNX object	EIBOBJECT
dpadCamera	IP camera	CAMERA
dpadScriptRunner	Run-script	CSCMD
dpadLoadControl	Load control	CSCMD

13.3 NOTES